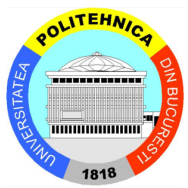

Session 0x0C
Return Oriented Programming (part 2)

Security Summer School

ACS/Ixia/Hexcellents

Recap

- ROP is like a ransom note
- We execute almost arbitrary code without writing any new code (aka shellcode)
- Requires much more work

Calling conventions

- ROP is mainly about setting up registers, adjusting the stack and calling functions
- We need to know:
  - How GCC compiles function calls
  - How the kernel expects syscalls to be set up

32 bit functions

Code editor

```
1  int function(void * arg1, void * arg2, void * arg3, void * arg4, void * arg5)
2  {
3    return 0;
4  }
5
6
7
8  int test(void * arg1, void * arg2, void * arg3, void * arg4, void * arg5){
9      function((void*)1, (void*)2, (void*)3, (void*)4, (void*)5);
10 }
```

Assembly output

```
1  function(void*, void*, void*, void*, void*):
2          push    ebp
3          mov     ebp, esp
4          mov     eax, 0
5          pop     ebp
6          ret
7  test(void*, void*, void*, void*, void*):
8          push    ebp
9          mov     ebp, esp
10         push    5
11         push    4
12         push    3
13         push    2
14         push    1
15         call    function(void*, void*, void*, void*, void*)
16         add     esp, 20
17         nop
18         leave
19         ret
20
```

32 bit syscalls

```
#  long int syscall (long int __sysno, ...)

gdb-peda$ pdis syscall
Dump of assembler code for function syscall:
   0x000e39e0 <+0>:         push   ebp
   0x000e39e1 <+1>:         push   edi
   0x000e39e2 <+2>:         push   esi
   0x000e39e3 <+3>:         push   ebx
   0x000e39e4 <+4>:         mov    ebp,DWORD PTR [esp+0x2c]
   0x000e39e8 <+8>:         mov    edi,DWORD PTR [esp+0x28]
   0x000e39ec <+12>:         mov    esi,DWORD PTR [esp+0x24]
   0x000e39f0 <+16>:         mov    edx,DWORD PTR [esp+0x20]
   0x000e39f4 <+20>:         mov    ecx,DWORD PTR [esp+0x1c]
   0x000e39f8 <+24>:         mov    ebx,DWORD PTR [esp+0x18]
   0x000e39fc <+28>:         mov    eax,DWORD PTR [esp+0x14]
   0x000e3a00 <+32>:         call   DWORD PTR gs:0x10
```

64 bit functions

Code editor

```
1  int function(void * arg1, void * arg2, void * arg3, void * arg4, void * arg5)
2  {
3    return 0;
4  }
5
6
7
8  int test(void * arg1, void * arg2, void * arg3, void * arg4, void * arg5){
9      function((void*)1, (void*)2, (void*)3, (void*)4, (void*)5);
10 }
```

Assembly output

```
1  function(void*, void*, void*, void*, void*):
2          push    rbp
3          mov     rbp, rsp
4          mov     QWORD PTR [rbp-8], rdi
5          mov     QWORD PTR [rbp-16], rsi
6          mov     QWORD PTR [rbp-24], rdx
7          mov     QWORD PTR [rbp-32], rcx
8          mov     QWORD PTR [rbp-40], r8
9          mov     eax, 0
10         pop     rbp
11         ret
12 test(void*, void*, void*, void*, void*):
13         push    rbp
14         mov     rbp, rsp
15         sub     rsp, 40
16         mov     QWORD PTR [rbp-8], rdi
17         mov     QWORD PTR [rbp-16], rsi
18         mov     QWORD PTR [rbp-24], rdx
19         mov     QWORD PTR [rbp-32], rcx
20         mov     QWORD PTR [rbp-40], r8
21         mov     r8d, 5
22         mov     ecx, 4
23         mov     edx, 3
24         mov     esi, 2
25         mov     edi, 1
26         call    function(void*, void*, void*, void*, void*)
27         nop
28         leave
29         ret
```

64 bit syscalls

```
#  long int syscall (long int __sysno, ...)

gdb-peda$ pdis syscall
Dump of assembler code for function syscall:
   0x00000000000e4ac0 <+0>:         mov    rax,rdi
   0x00000000000e4ac3 <+3>:         mov    rdi,rsi
   0x00000000000e4ac6 <+6>:         mov    rsi,rdx
   0x00000000000e4ac9 <+9>:         mov    rdx,rcx
   0x00000000000e4acc <+12>:         mov    r10,r8
   0x00000000000e4acf <+15>:         mov    r8,r9
   0x00000000000e4ad2 <+18>:         mov    r9,QWORD PTR [rsp+0x8]
   0x00000000000e4ad7 <+23>:         syscall
```

Techniques in this session

- Handling ASLR with a ROP-based information leak
- Stack space fixing: when the initial overflow is not enough for the entire payload
- Hybrid exploits: using ROP to create a RWX page and then executing shellcode in that region
- Syscalls using ROP

Exploit automation

- Writing exploits in bash is error-prone and only allows static payloads
- Because of ASLR static payloads are useless. Use Python!
- We need something to facilitate I/O with the vulnerable binary: either locally or remotely

Exploit automation

- Writing exploits in bash is error-prone and only allows static payloads
- Because of ASLR static payloads are useless. Use Python!
- We need something to facilitate I/O with the vulnerable binary: either locally or remotely

PWNTOOLS

Task 0 walkthrough

Pwntools demo time!