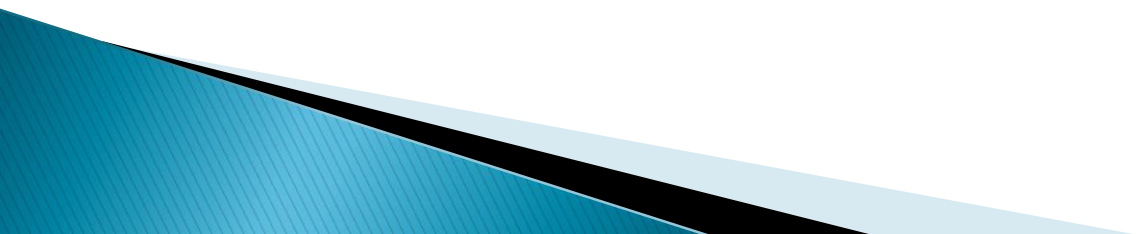
Binary Exploitation 2
Return Oriented Programming



Where Are We ?

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.
- Non Executable, No Canaries and No ASLR.
  - Overwrite the return address.
  - Reurn to libc restricted by system().

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.
- Non Executable, No Canaries and No ASLR.
  - Overwrite the return address.
  - Reurn to libc restricted by system().
- Non Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Return Oriented Programming.
  - Execute arbitrary code.

Return Oriented Programming Attacks

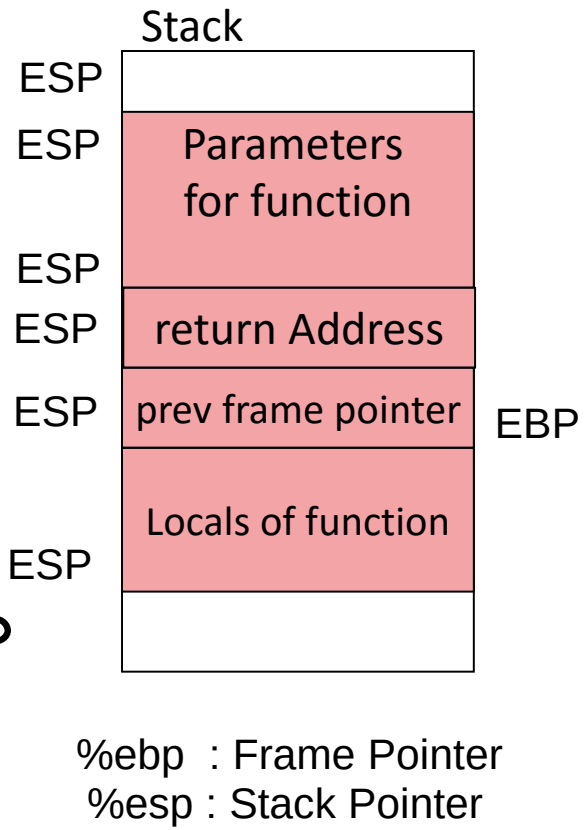
- Discovered by Hovav Shacham of Stanford University
- Subverts execution.
  - As with the regular ret-2-libc, can be used with non executable stacks since the instructions can be legally executed.
  - Unlike ret-2-libc does not require to execute functions in libc (can execute any arbitrary code).

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls on the x86

Stack : Function Call

Call instruction has 2 steps:

- Push the contents pointed to by EIP.
- Decrease ESP by 4 (32bit machine)



In main

```
push $3
push $2
push $1
```

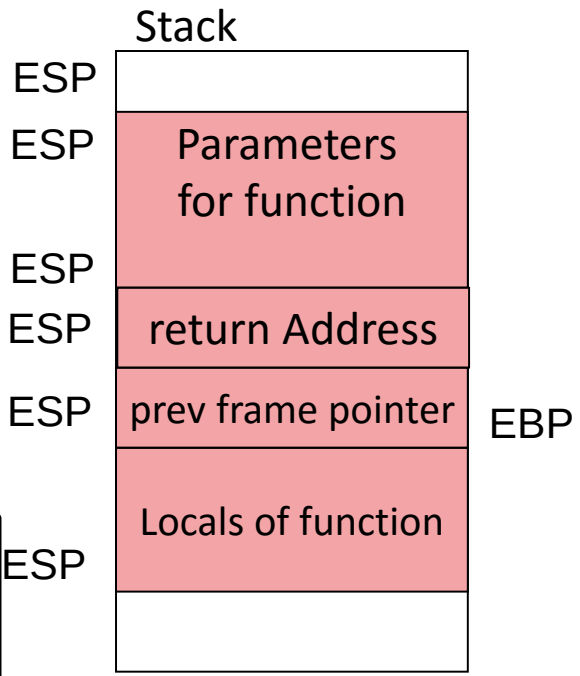
In function

```
push %ebp
movl %esp, %ebp
sub $20, %esp
```

Stack : Function Return

Ret instruction has 2 steps:

- Pops the contents pointed to by ESP into EIP
- Increment ESP by 4 (32bit machine)



In main

```
push $3
push $2
push $1
```

In function

```
movl %ebp, %esp
```

Action by leave instruction

%ebp : Frame Pointer
%esp : Stack Pointer

Target Payload

Lets say this is the payload needed to be executed by an attacker.

```
"movl %esi, 0x8(%esi);"
"movb $0x0, 0x7(%esi);"
"movl $0x0, 0xc(%esi);"
"movl $0xb, %eax;"
"movl %esi, %ebx;"
"leal 0x8(%esi), %ecx;"
"leal 0xc(%esi), %edx;"
```

Suppose there is a function in libc, which has exactly this sequence of instructions … then we are done.. we just need to subvert execution to the function

Target Payload

Lets say this is the payload needed to be executed by an attacker.

```
"movl %esi, 0x8(%esi);"
"movb $0x0, 0x7(%esi);"
"movl $0x0, 0xc(%esi);"
"movl $0xb, %eax;"
"movl %esi, %ebx;"
"leal 0x8(%esi), %ecx;"
"leal 0xc(%esi), %edx;"
```

Suppose there is a function in libc, which has exactly this sequence of instructions … then we are done.. we just need to subvert execution to the function

What if such a function does not exist?

If you can't find it then build it

Step 1: Find Gadgets

- Find gadgets.
- A gadget is a short sequence of instructions followed by a return.

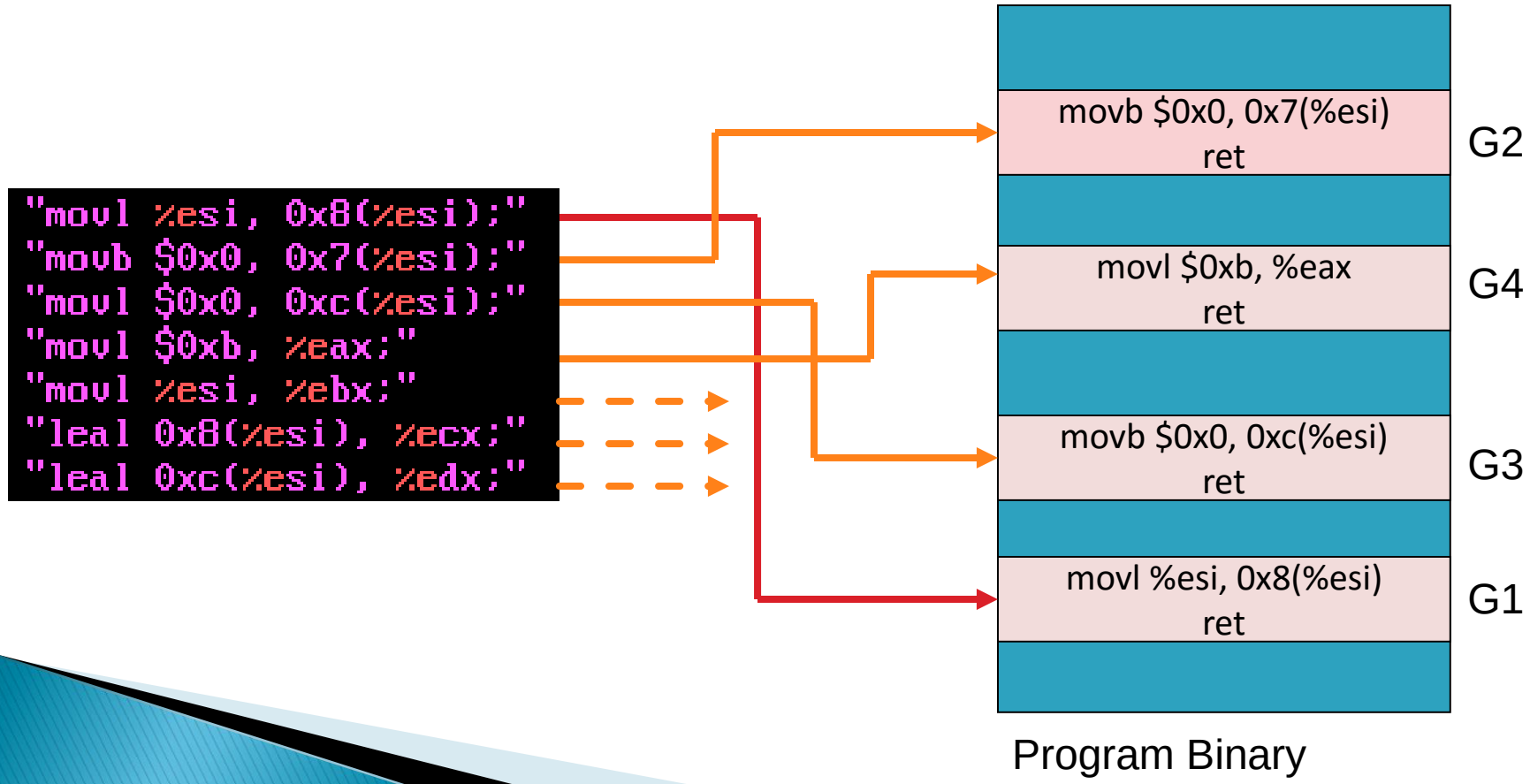
```
useful instruction(s)
ret
```

- Useful instructions : should not transfer control outside the gadget.

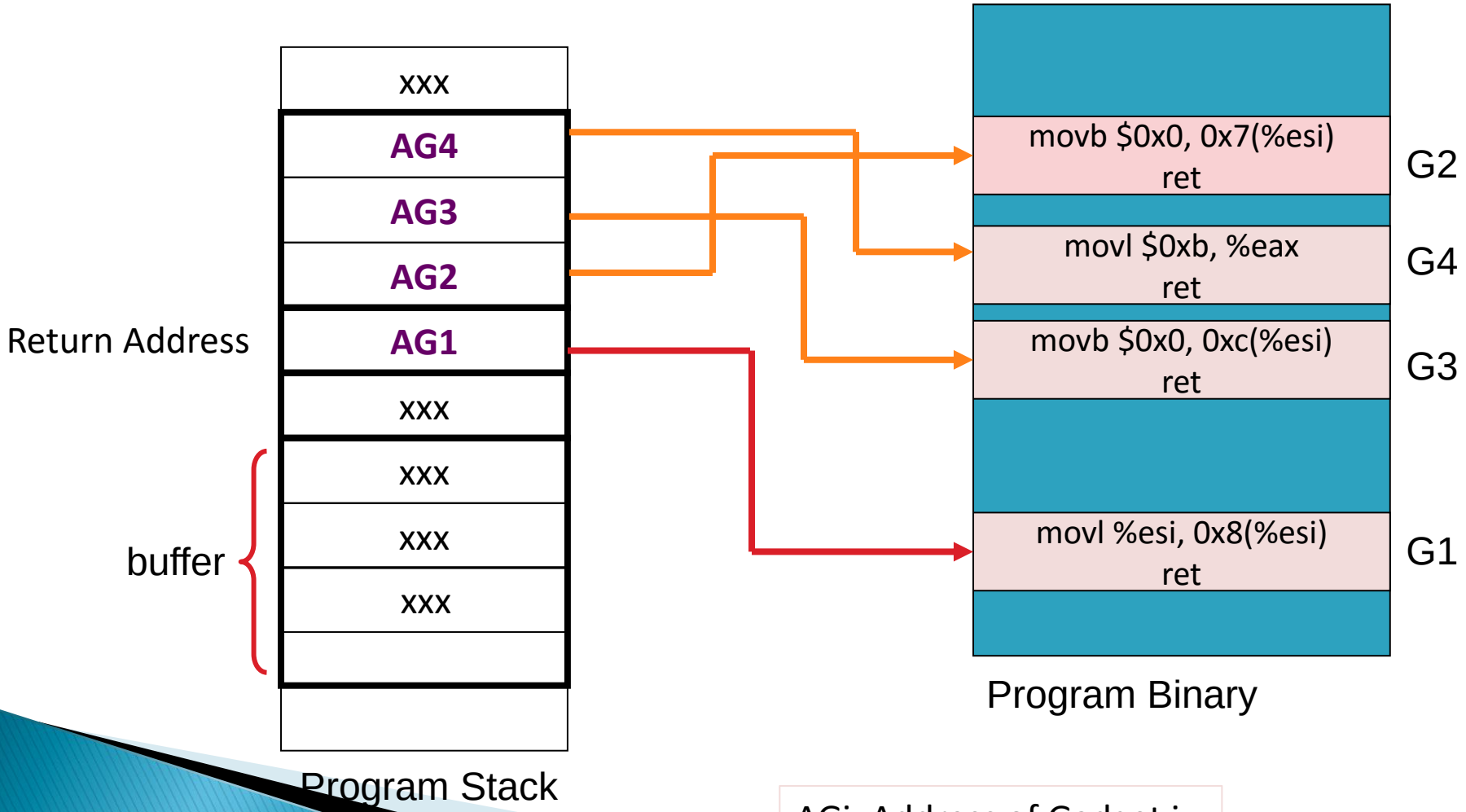
- This is a pre-processing step by statically analysing the libc library.

Step 2: Stitching

- Stitch gadgets so that the payload is built



Step 3: Construct the Stack



AGi: Address of Gadget i

Finding Gadgets

- Static analysis of libc
- To find
  1. A set of instructions that end in a ret (0xc3).
     The instructions can be intended (put in by the compiler) or unintended.
  2. Besides ret, none of the instructions transfer control out of the gadget.

Intended vs Unintended Instructions

- **Intended :** machine code intentionally put in by the compiler
- **Unintended :** interpret machine code differently in order to build new instructions

Machine Code :	F7 C7 07 00 00 00 0F 95 45 C3

What the compiler intended..

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

What was not intended

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

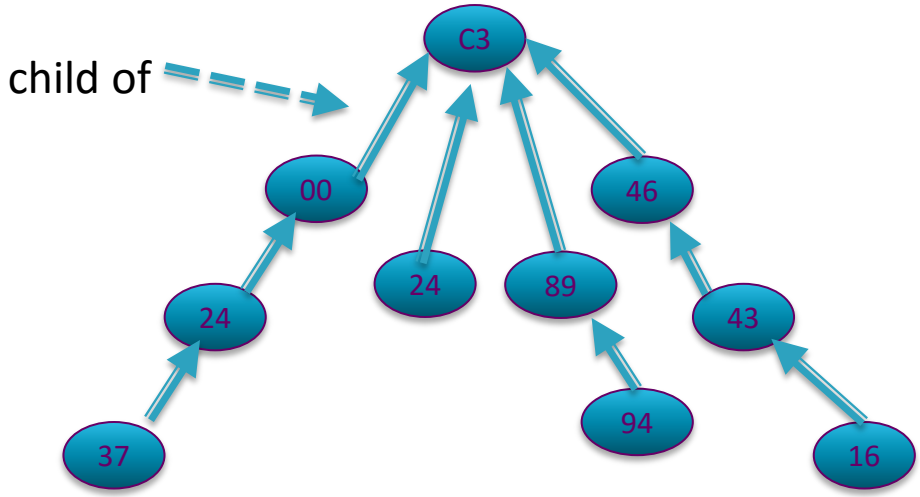
Highly likely to find many diverse instructions of this form in x86.
Not so likely to have such diverse instructions in RISC processors.

Geometry

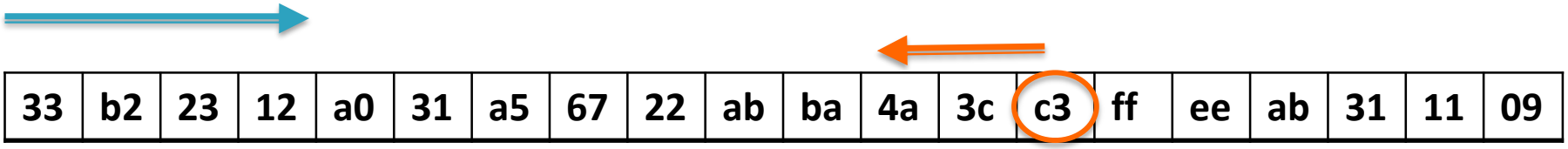
- Given an arbitrary string of machine code, what is the probability that the code can be interpreted as useful instructions.
  - x86 code is highly dense.
  - RISC processors like (SPARC, ARM, etc.) have low geometry.
- Thus finding gadgets in x86 code is considerably more easier than that of ARM or SPARC.
- Fixed length instruction set reduces geometry.

Finding Gadgets

- Static analysis of libc.
- Find any memory location with 0xc3 (RET instruction).
- Build a trie data structure with 0xc3 as a root.
- Every path from leaf to the root is a possible gadget.



Finding Gadgets



- Scan libc from the beginning toward the end
- If 0xc3 is found
  - Start scanning backward
  - With each byte, ask the question if the subsequence forms a valid instruction
  - If yes, add as child
  - If no, go backwards until we reach the maximum instruction length (20 bytes)
  - Repeat this till (a predefined) length W, which is the max instructions in the gadget

Finding Gadgets Algorithm

Algorithm GALILEO:
create a node, *root*, representing the ret instruction;
place *root* in the trie;
for *pos* **from** 1 **to** *textseg_len* **do**:
if the byte at *pos* is c3, i.e., a ret instruction, **then**:
call BUILDFROM(*pos*, *root*).

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
for *step* **from** 1 **to** *max_insn_len* **do**:
if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
ensure *insn* is in the trie as a child of *parent_insn*;
if *insn* isn't boring **then**:
call BUILDFROM(*pos* − *step*, *insn*).

Finding Gadgets Algorithm

Algorithm GALILEO:
create a node, *root*, representing the ret instruction;
place *root* in the trie;
for *pos* **from** 1 **to** *textseg_len* **do**:
if the byte at *pos* is c3, i.e., a ret instruction, **then**:
call BUILDFROM(*pos*, *root*).

Found 15,121 nodes in ~1MB of libc binary

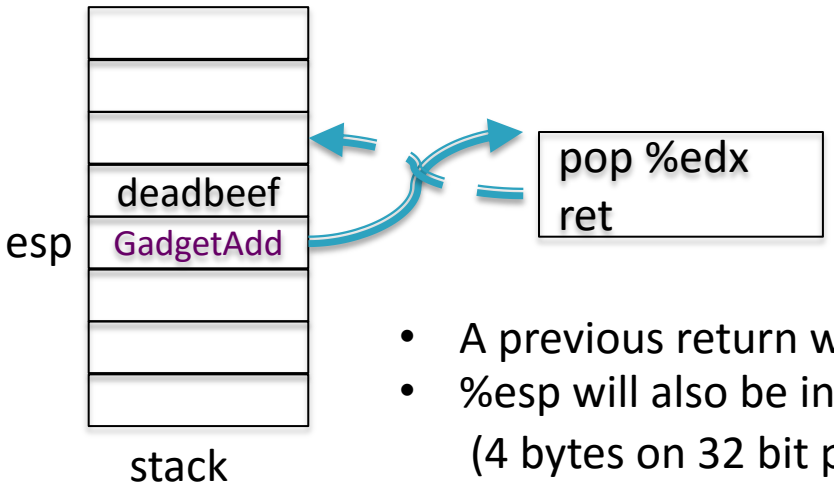
is this sequence of instructions valid x86 instruction?

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
for *step* **from** 1 **to** *max_insn_len* **do**:
if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
ensure *insn* is in the trie as a child of *parent_insn*;
if *insn* isn't boring **then**:
call BUILDFROM(*pos* − *step*, *insn*).

Boring: not interesting to look further;
Eg. pop %ebp; ret;;;; leave; ret (these are boring if we want to ignore intended instructions)
Jump out of the gadget instructions

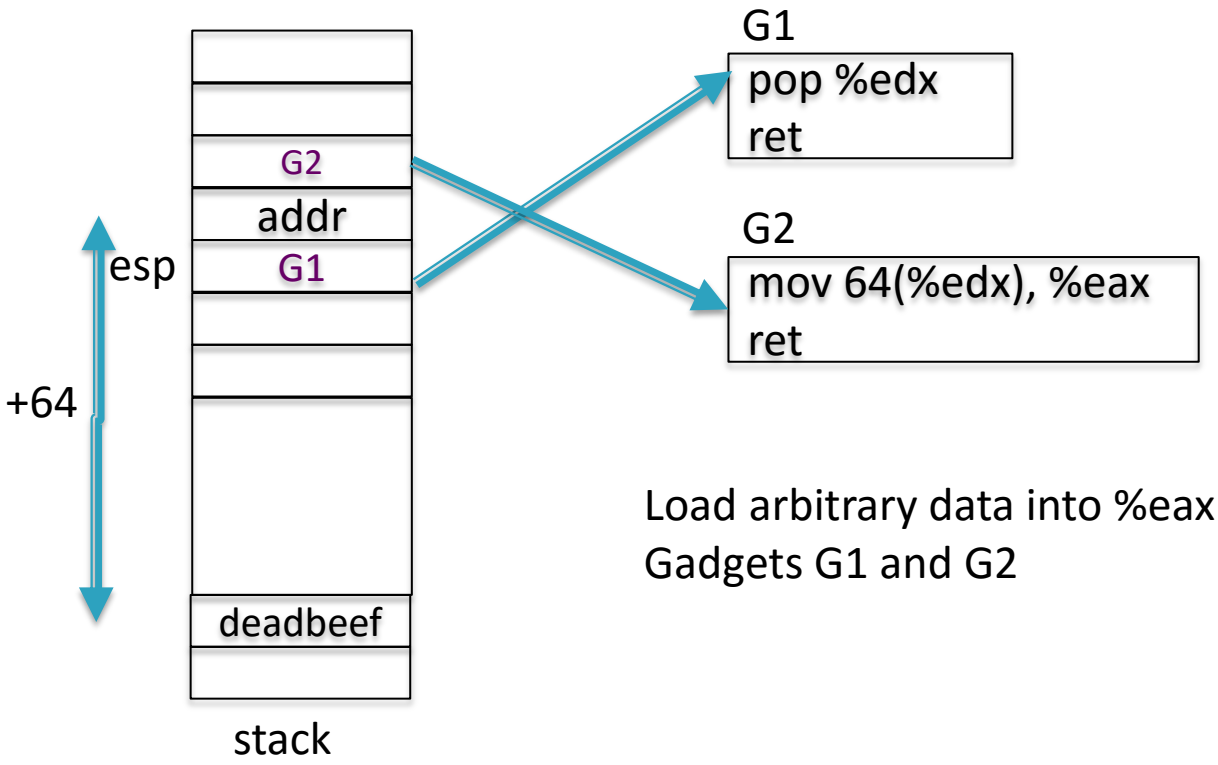
Gadgets : Constant into Register

Loading a constant into a register (edx ← deadbeef)



- A previous return will pop the gadget address into %eip
- %esp will also be incremented to point to deadbeef (4 bytes on 32 bit platform)
- The pop %edx will pop deadbeef from the stack and increment %esp to point to the next 4 bytes on the stack

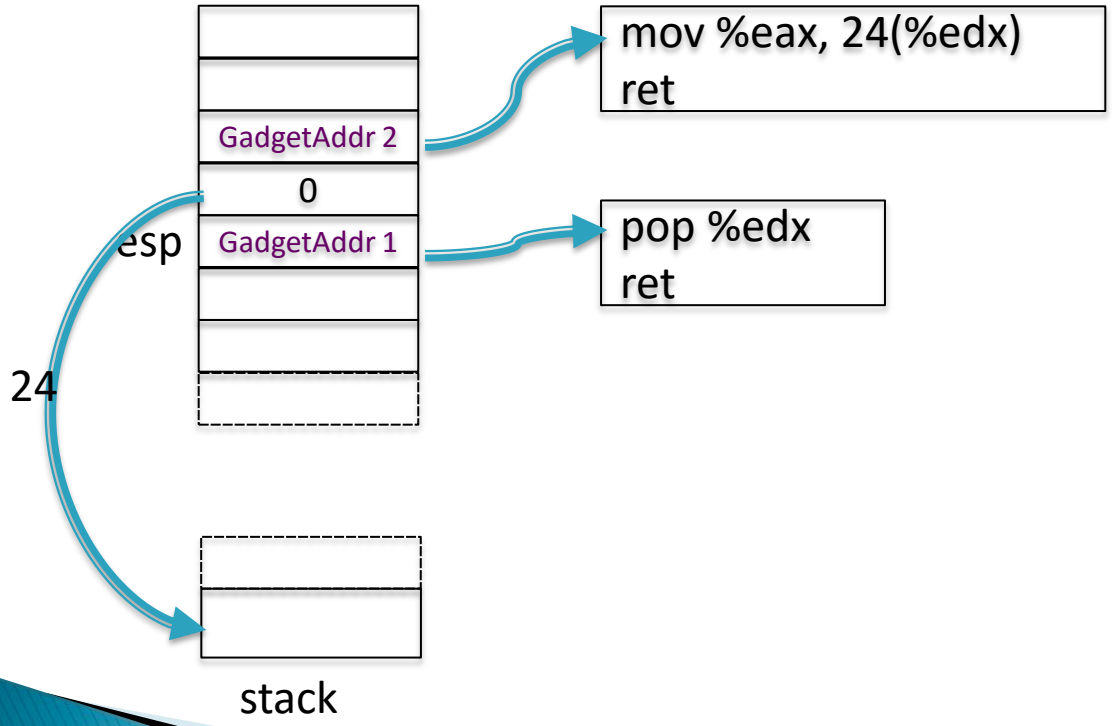
Gadgets : Arbitrary Data into eax



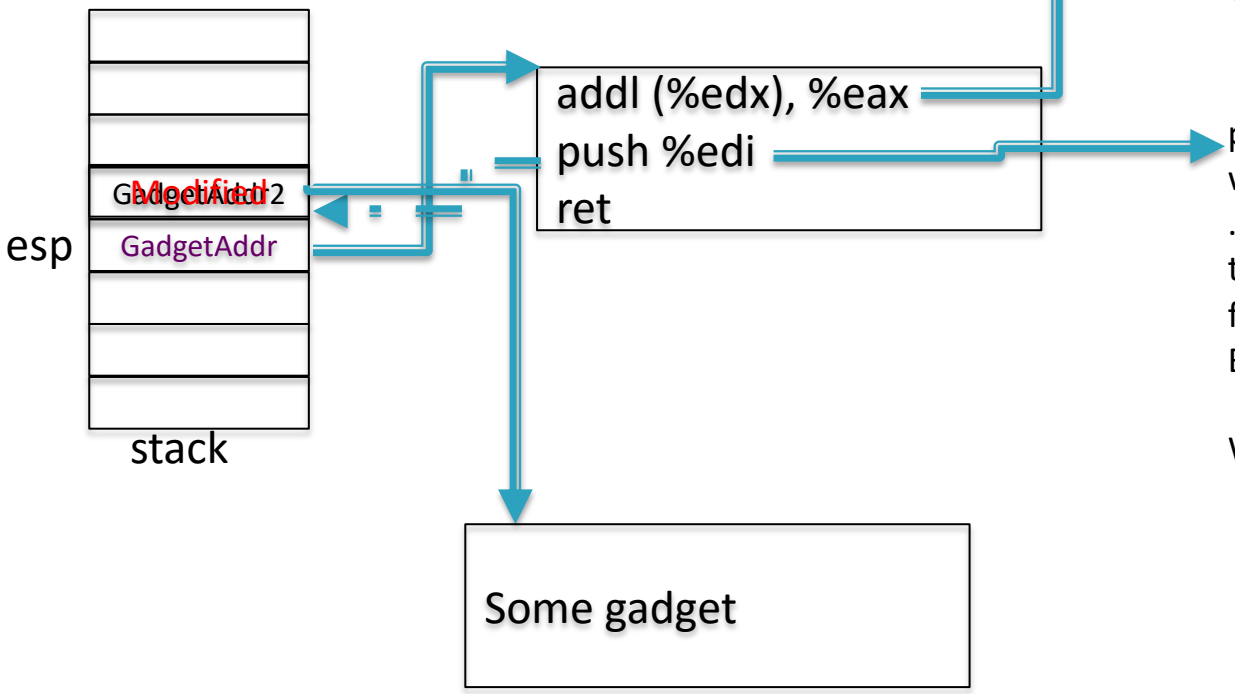
Load arbitrary data into %eax register using Gadgets G1 and G2

Gadgets: Store Constants

- Store the contents of a register to a memory location in the stack



Gadget: Addition

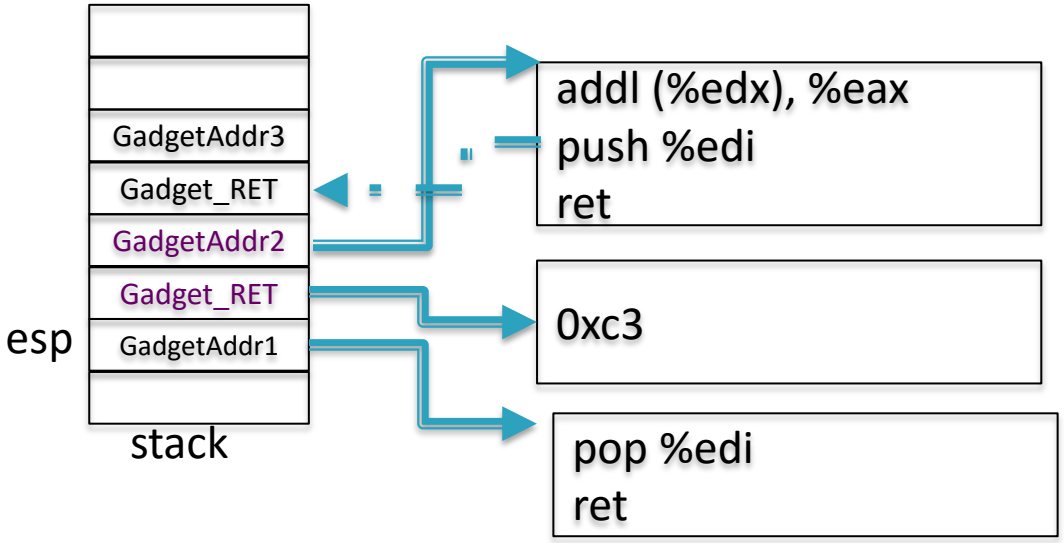


Add the memory pointed to by %edx to %eax.
The result is stored in %eax

pushes %edi.. onto the stack
why is this present?
…. This is unnecessary, but this is best gadget that we can find for addition
But can create problems!!

We need work arounds!

Gadgets: Addition with NOP

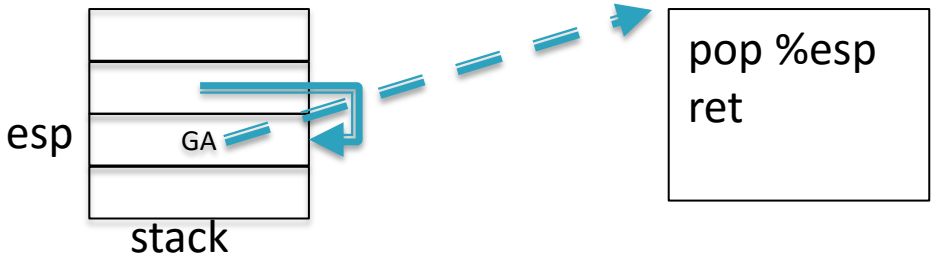


1. First put gadget ptr for 0xC3 into %edi
2. 0xC3 corresponds to NOP in ROP
3. push %edi in gadget 2 just pushes 0xc3 back into the stack
   Therefore not disturbing the stack contents
4. Gadget 3 executes as planned

0xc3 is ret in ROP and ret is equivalent to NOP instruction

Unconditional Branches

- Changing the %esp



Conditional Branches

In x86 instructions conditional branches have 2 parts.

1. An instruction which modifies a condition flag (eg CF, OF, ZF).
   eg. **CMP %eax, %ebx** (will set ZF if %eax = %ebx)
2. A branch instruction (eg. JZ, JCC, JNZ, etc).
   which internally checks the conditional flag and
   changes the EIP accordingly.

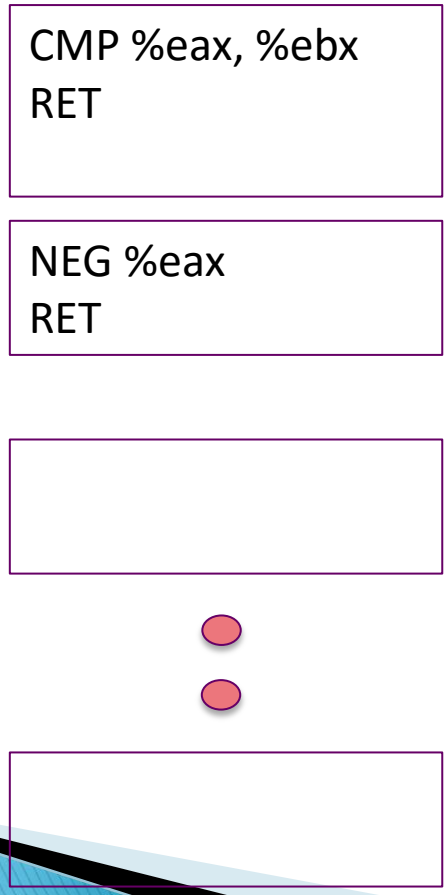
In ROP, we need flags to modify %esp register instead of EIP
Needs to be explicitly handled

In ROP conditional branches have 3 parts.

1. An ROP which modifies a condition flag (eg CF, OF, ZF).
   eg. CMP %eax, %ebx (will set ZF if %eax = %ebx)
2. Transfer flags to a register or memory.
3. Perturb %esp based on flags stored in memory.

Step 1 : Set the flags

Find suitable ROPs that set appropriate flags



subtraction
Affects flags CF, OF, SF, ZF, AF, PF

2s complement negation
Affects flags CF

Step 2: Transfer flags to memory or register

- Using **lahf** instruction
  stores 5 flags (ZF, SF, AF, PF, CF) in the %ah register

- Using **pushf** instruction
  pushes the eflags into the stack

where would one use this instruction?

ROPs for these two not easily found.

A third way – perform an operation whose result depends on the flag contents.

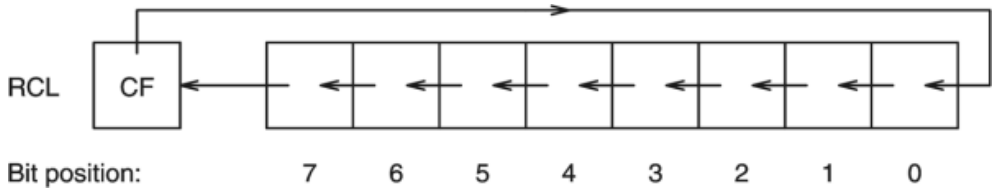
Step 2: Indirect way to transfer flags to memory

Several instructions operate using the contents of the flags

ADC %eax, %ebx : add with carry that performs eax <- eax + ebx + CF.

(if eax and ebx are 0 initially, then the result will be either 1 or 0 depending on the CF)

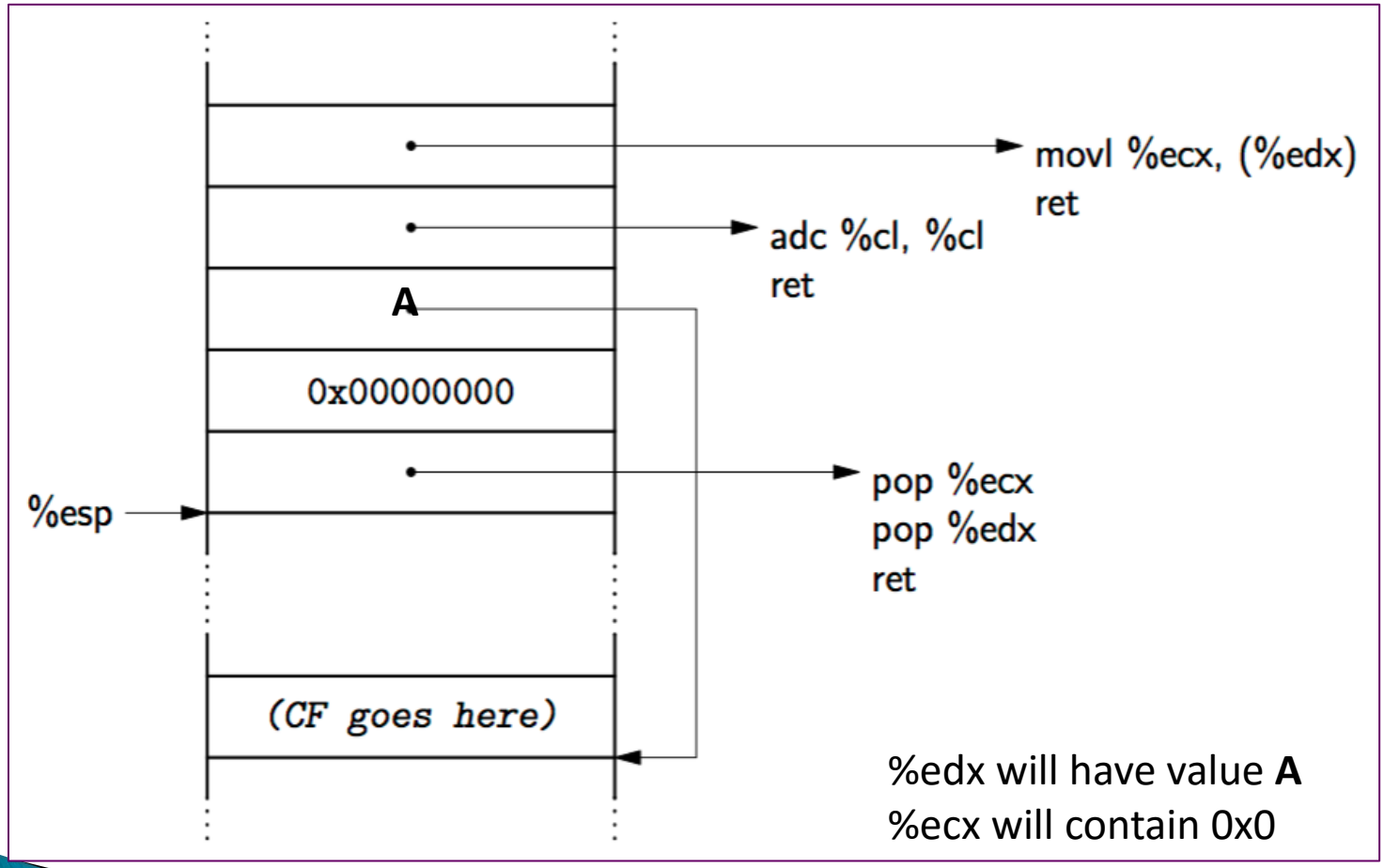
RCL : rotate left with carry.



RCL %eax, 1

(if eax = 0. then the result is either 0 or 1 depending on CF)

Gadgets: Transfer Flags to Memory



Step 3: Perturb %esp depending on flag

What we hope to achieve

```
If (CF is set){
    perturb %esp
}else{
    leave %esp as it is
}
```

What we have

* CF stored in a memory location (say X).
* Current %esp.
* Delta, how much to perturb %esp.

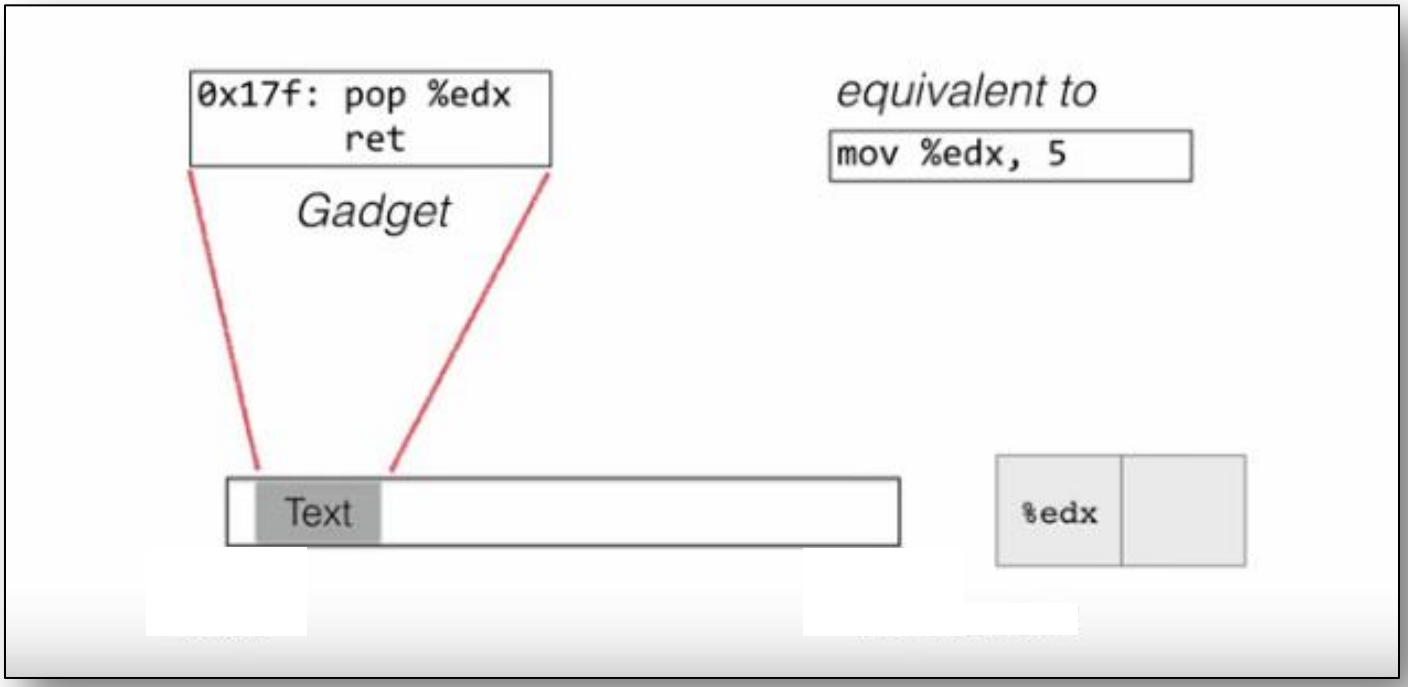
One way of achieving …

```
negate X
offset = Delta & X
%esp = %esp + offset
```

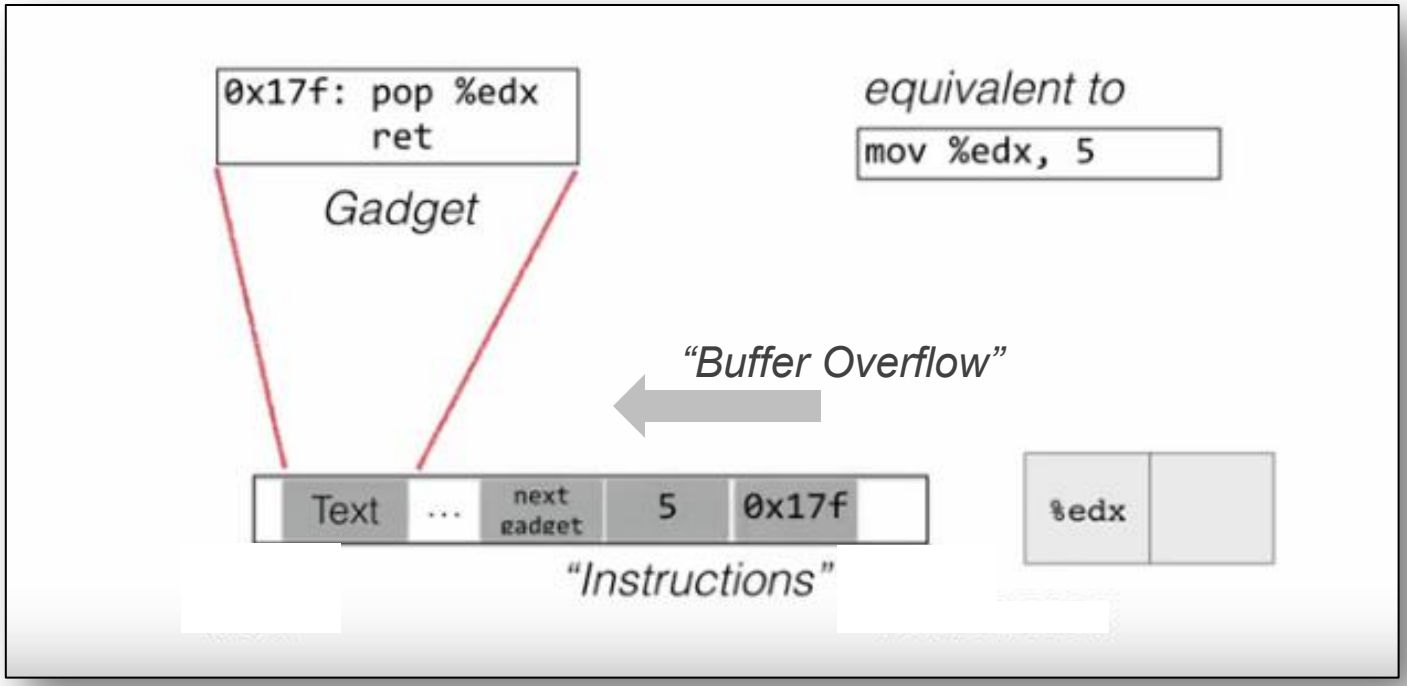
1. Negate X (eg. Using instruction negl)
   finds the 2's complement of X
   if (X = 1) 2's complement is 111111111…
   if (X = 0) 2's complement is 000000000...
2. offset = Delta if X = 1
   offset = 0 if X = 0
3. %esp = %esp + offset if X = 1
   %esp = %esp if X = 0

Gadgets: Example

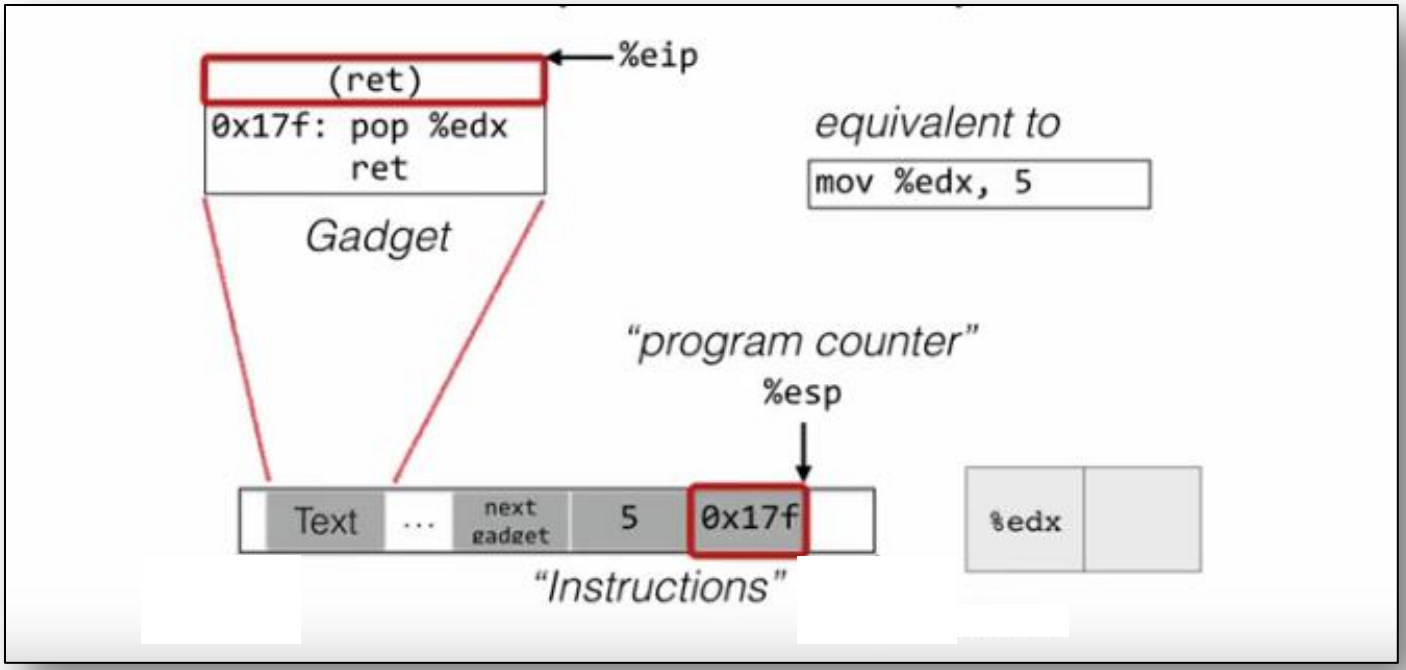
Gadgets: Example



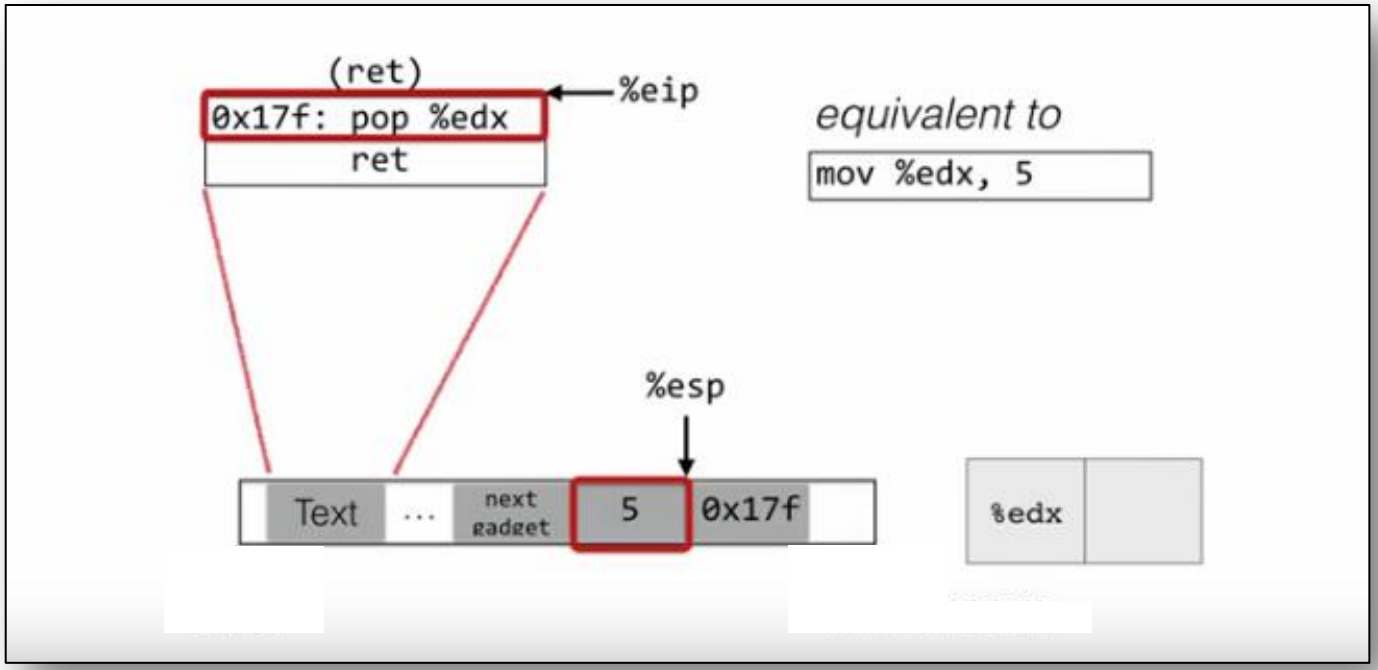
Gadgets: Example



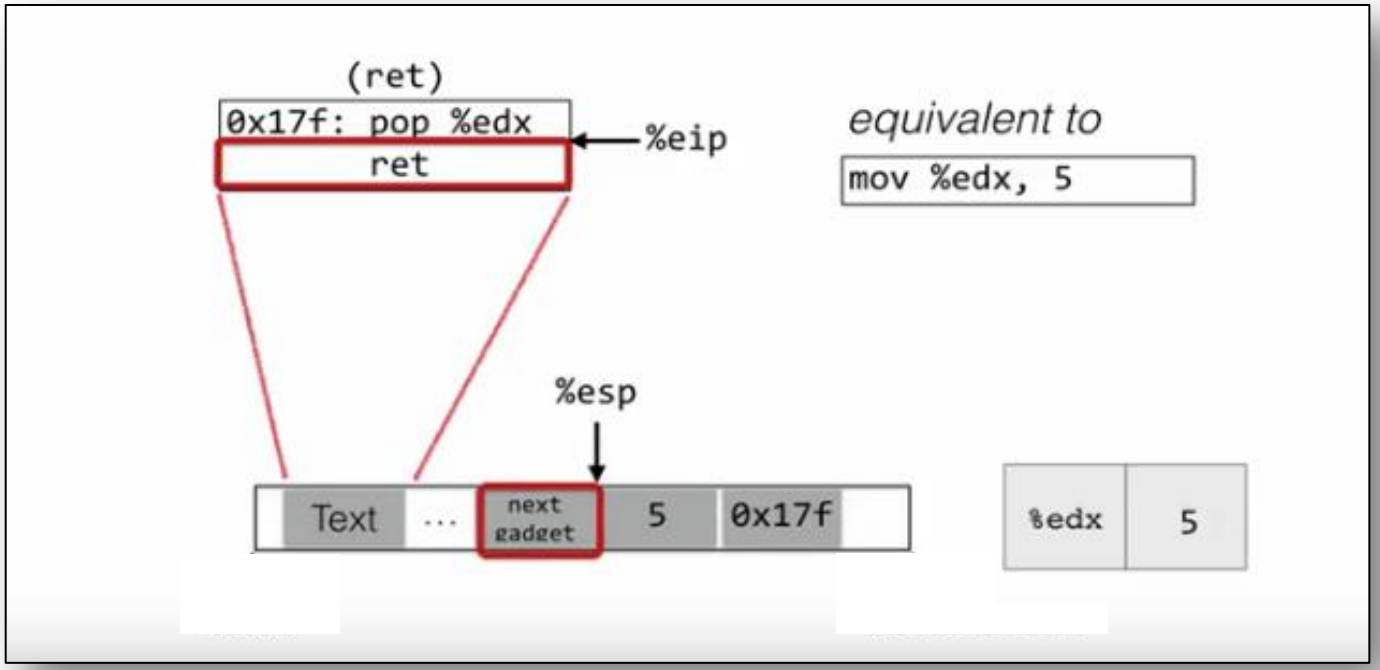
Gadgets: Example



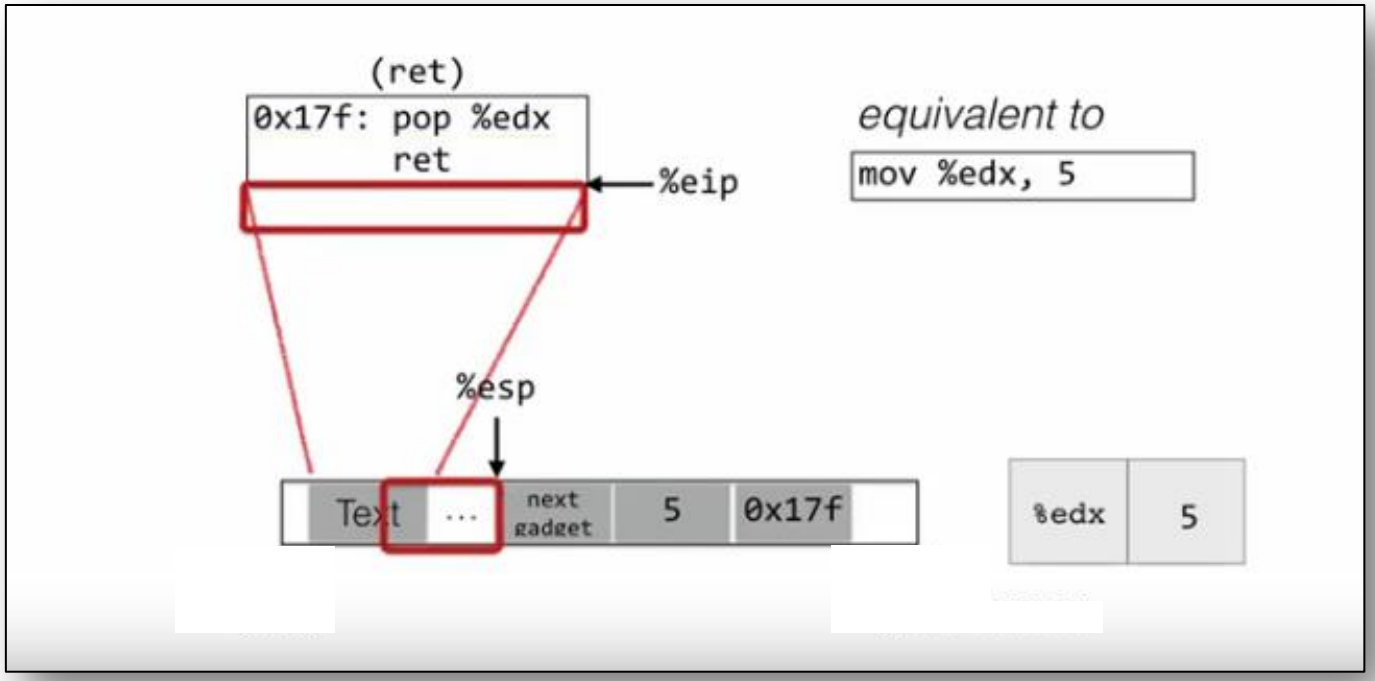
Gadgets: Example



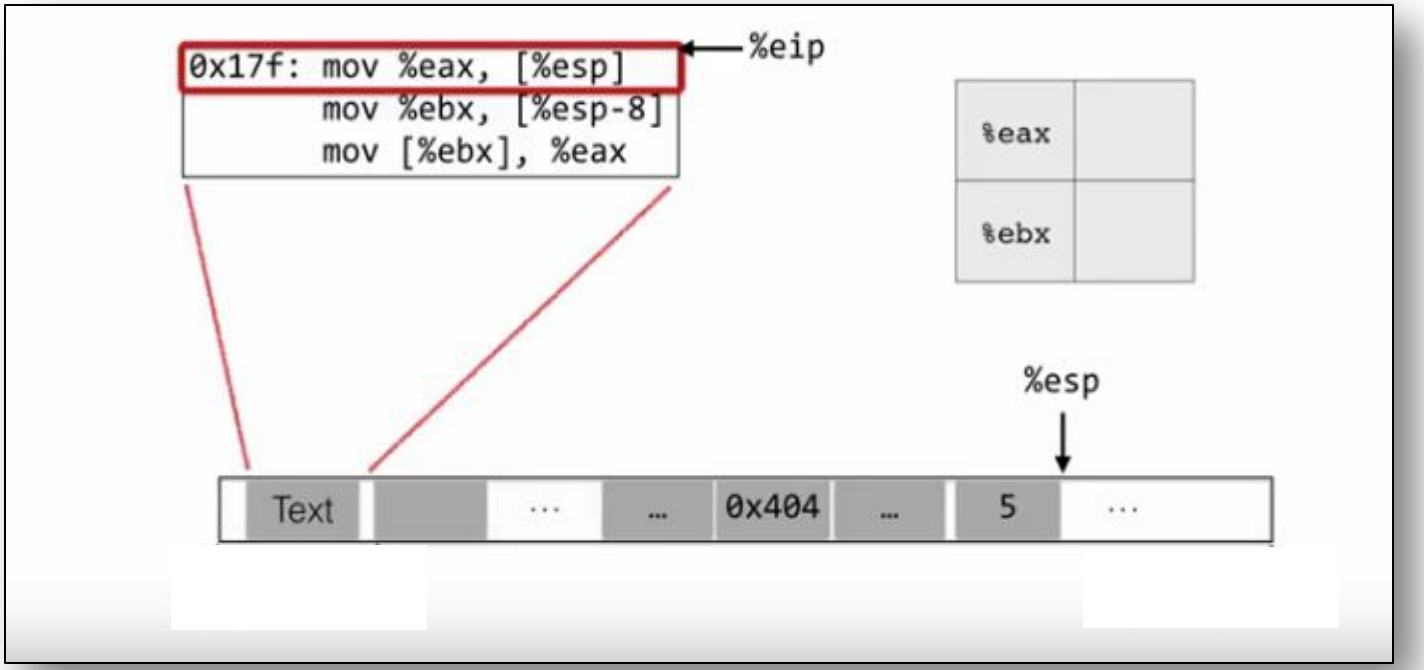
Gadgets: Example



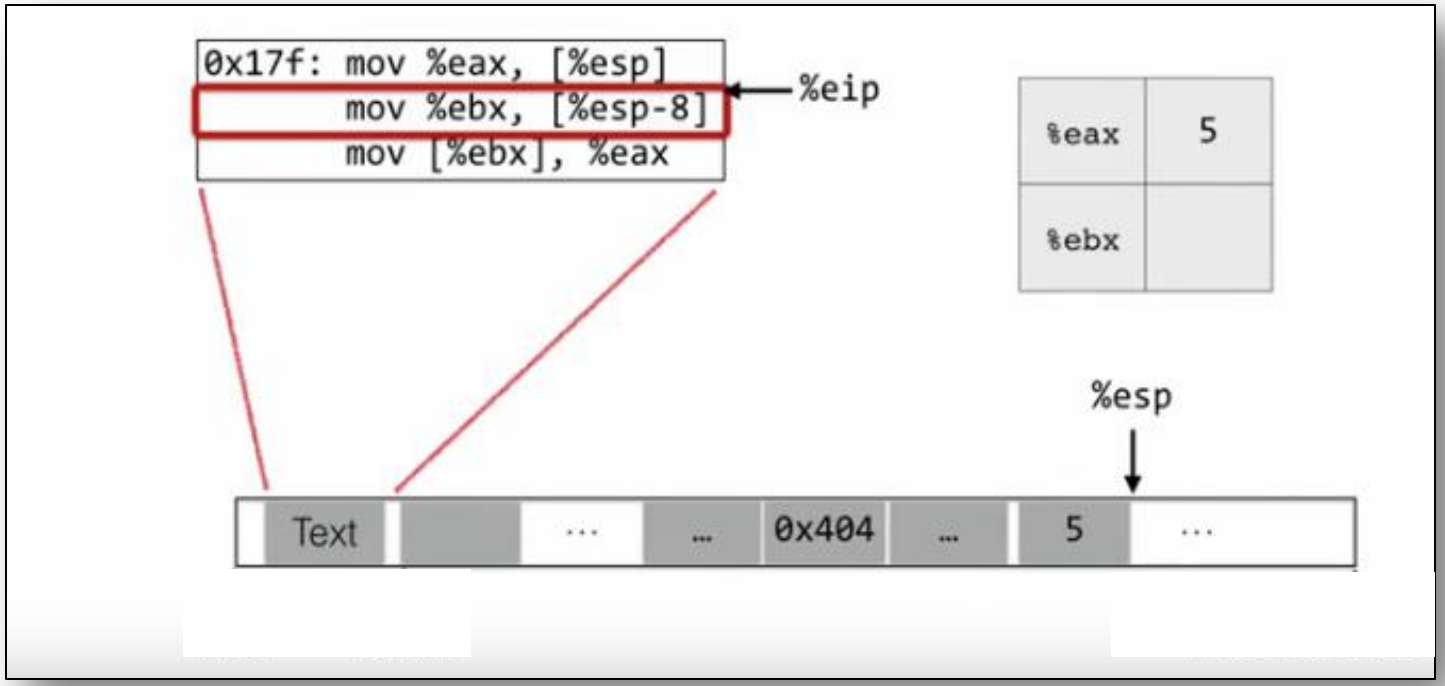
Gadgets: Example



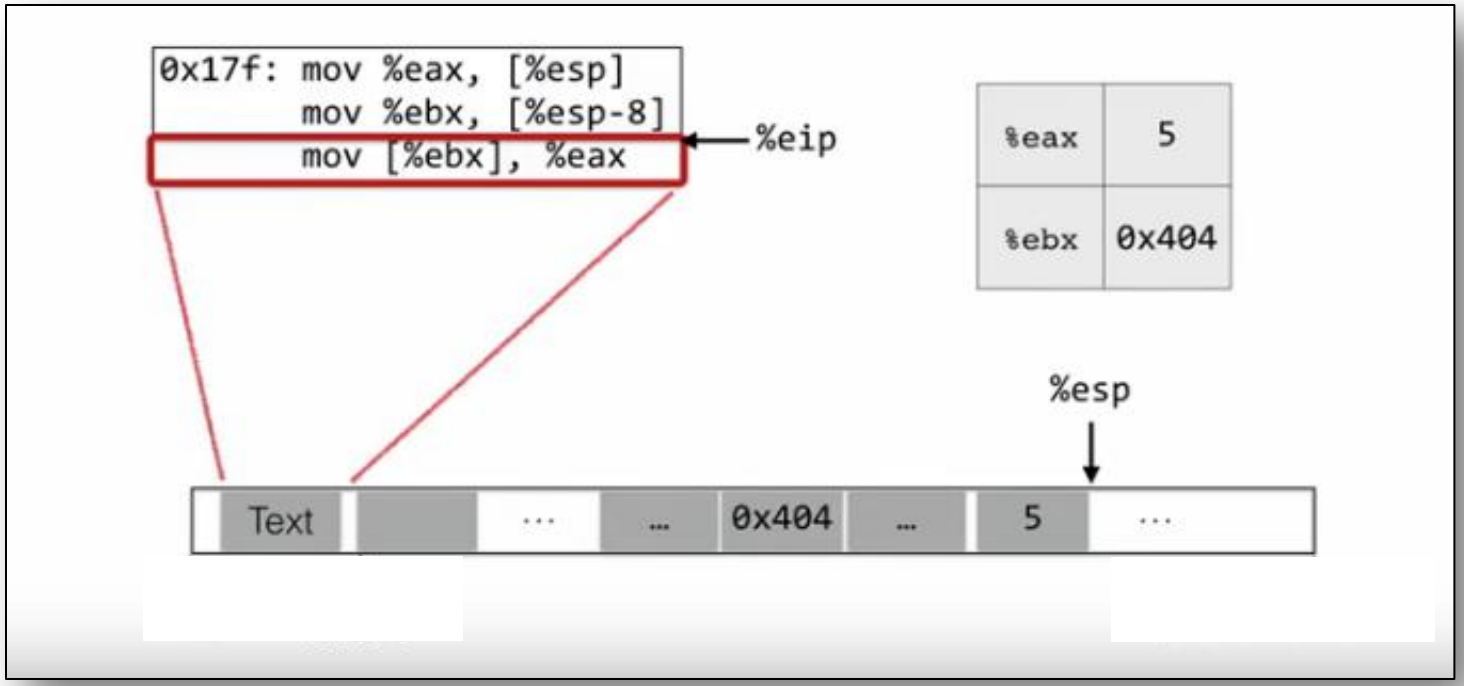
Gadgets: Another Example



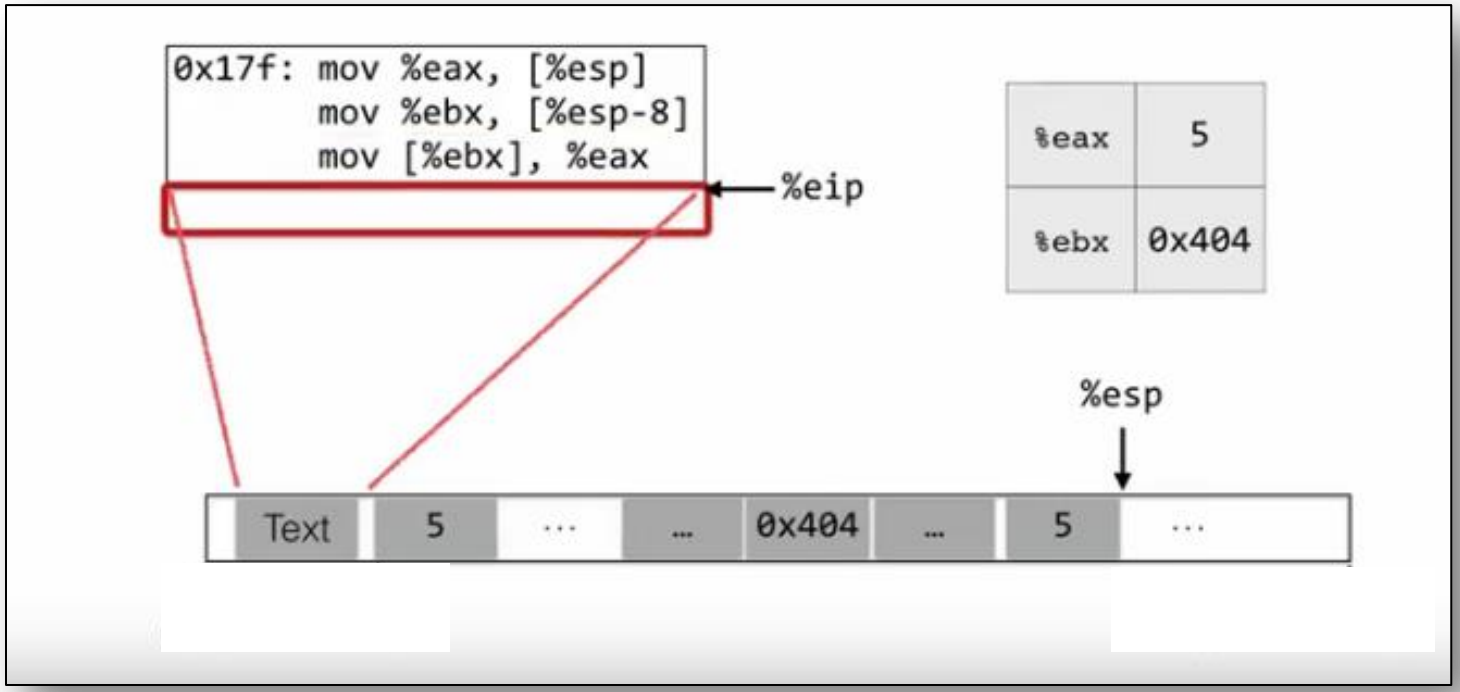
Gadgets: Another Example



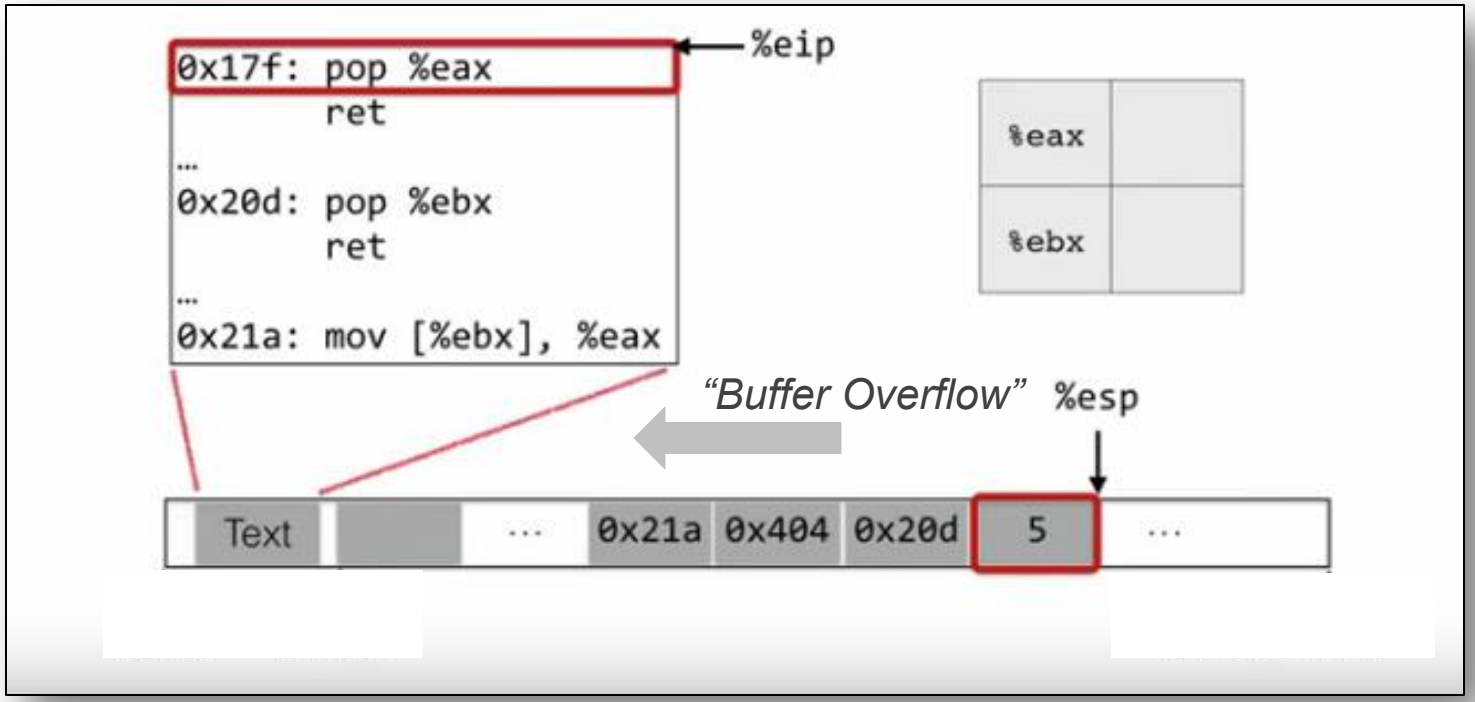
Gadgets: Another Example



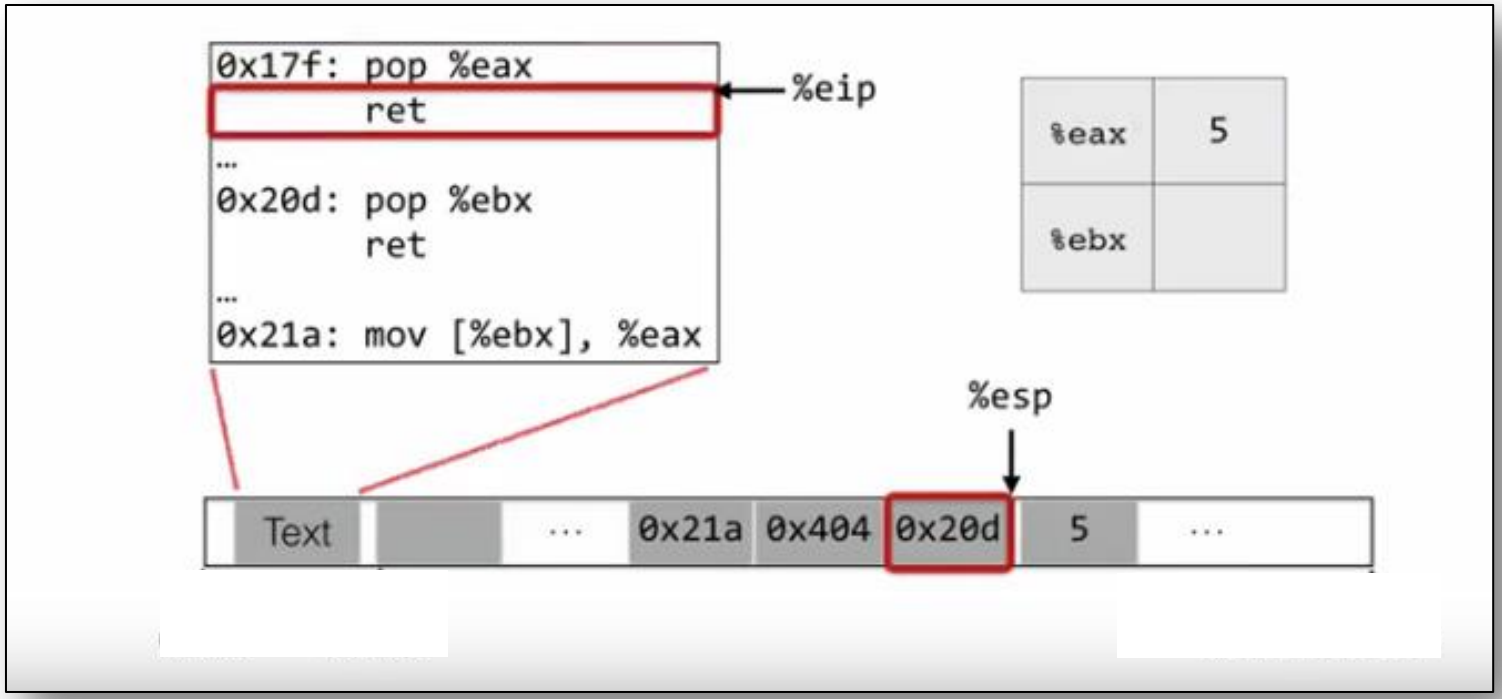
Gadgets: Another Example



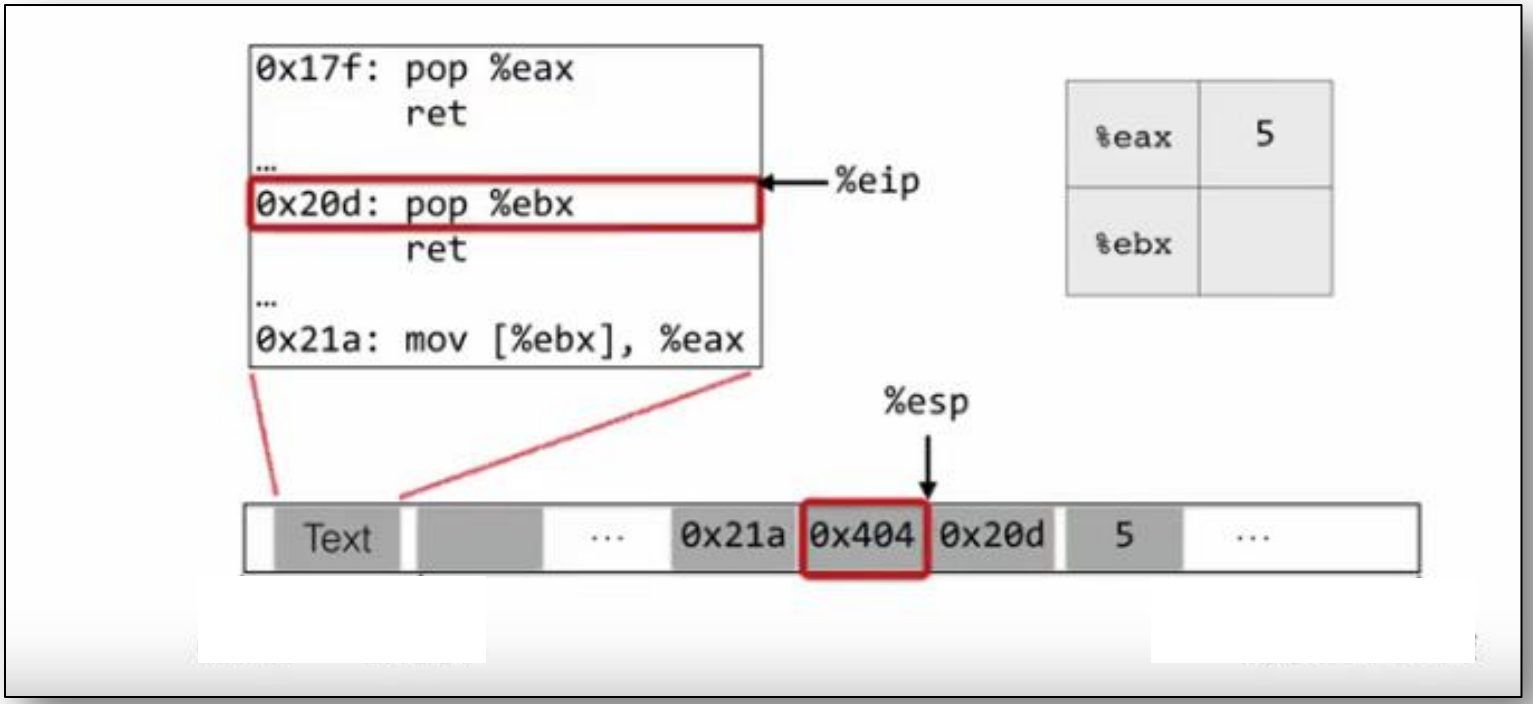
Gadgets: Another Example



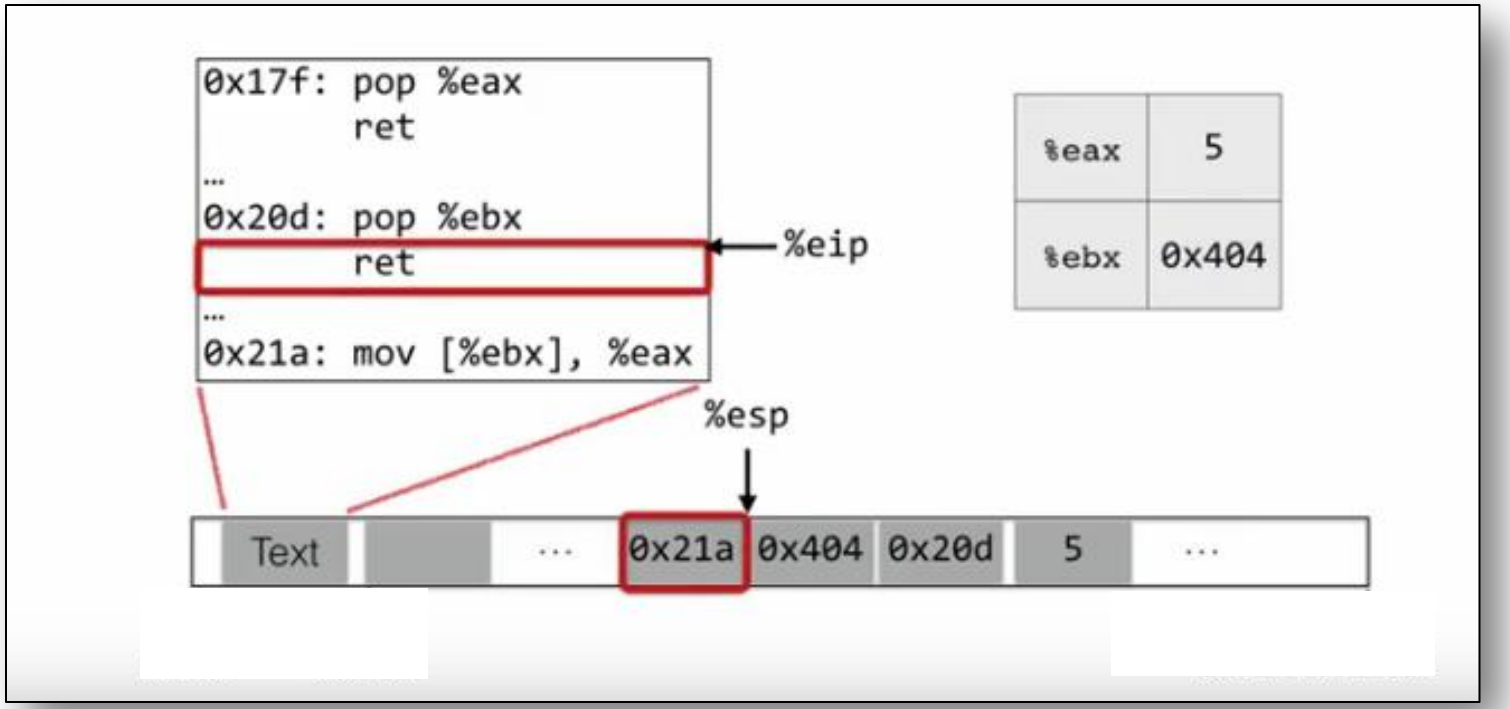
Gadgets: Another Example



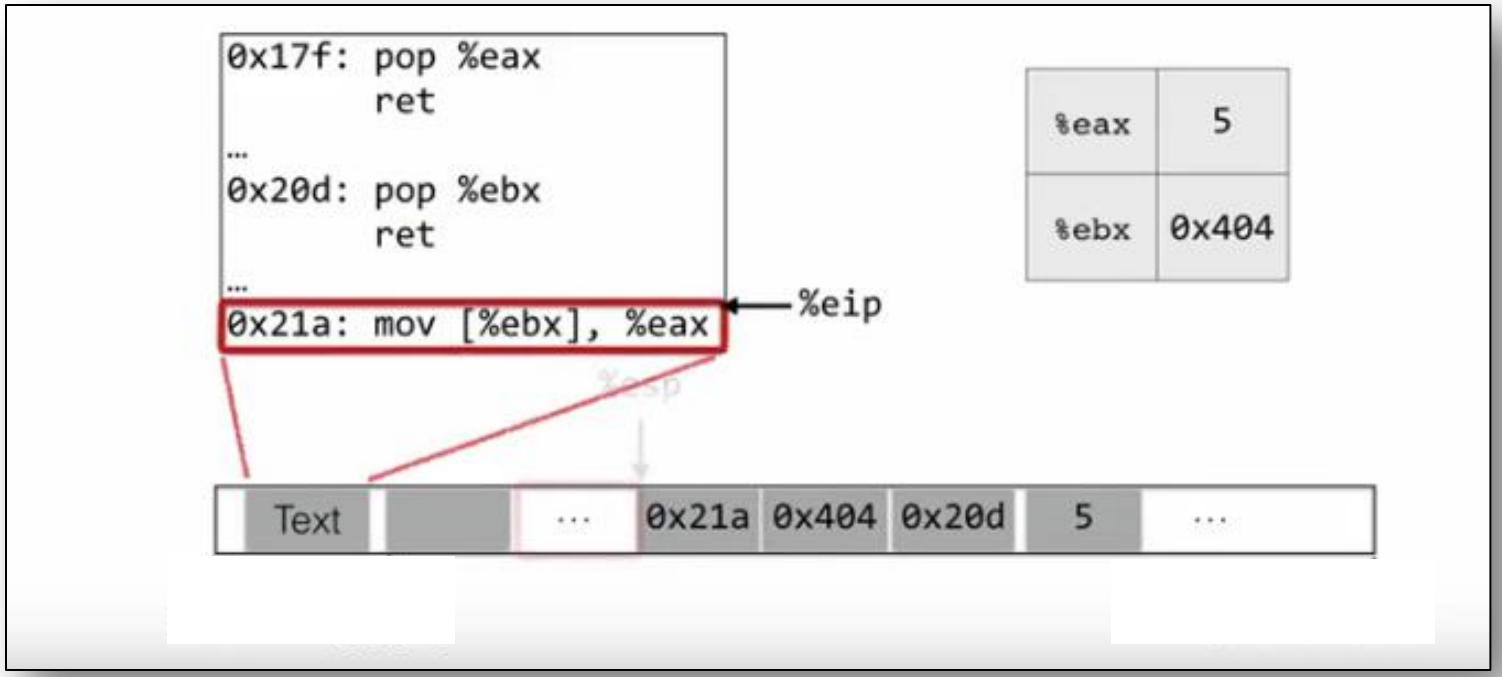
Gadgets: Another Example



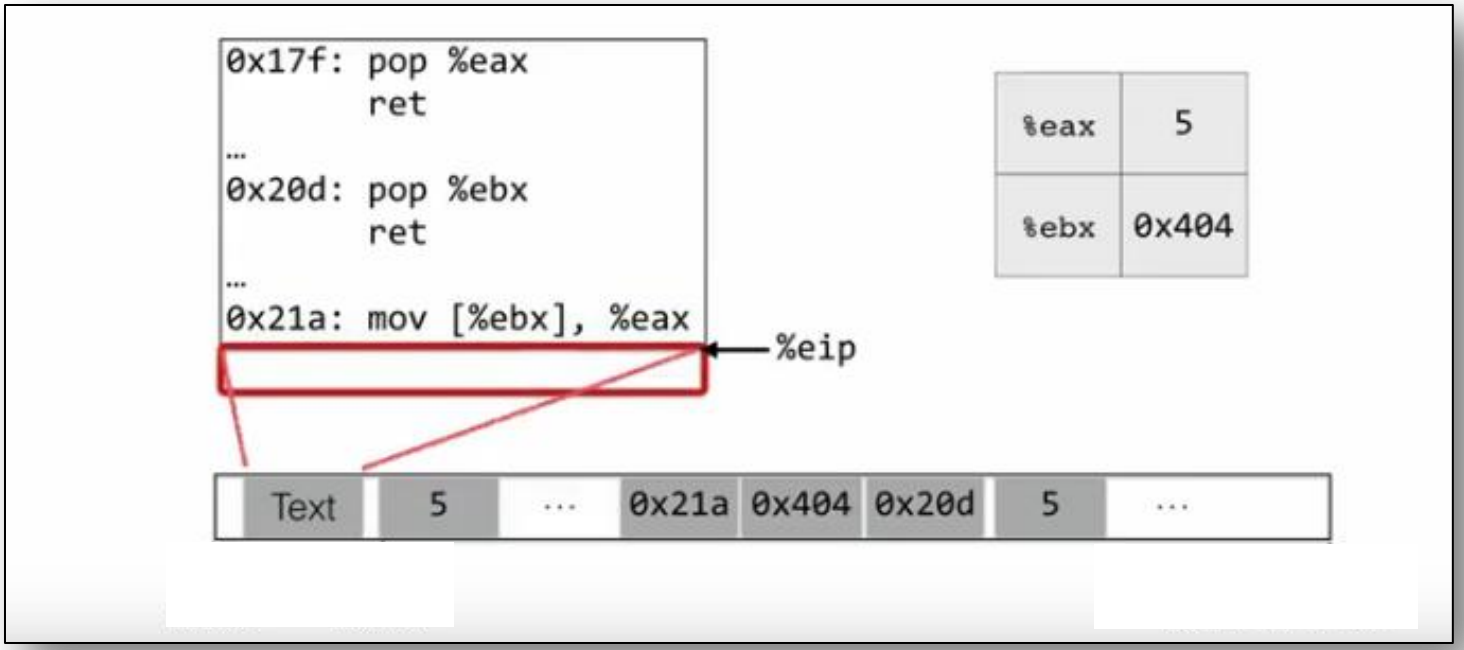
Gadgets: Another Example



Gadgets: Another Example



Gadgets: Another Example



Turing Complete

- Gadgets can do much more…
  invoke libc functions,
  invoke system calls, ...
- For x86, gadgets are said to be turning complete.
  - Can program just about anything with gadgets.
- For RISC processors, more difficult to find gadgets.
  - Instructions are fixed width.
  - Therefore can't find unintentional instructions.
- Tools available to find gadgets automatically.

  Eg. ROPGadget (https://github.com/JonathanSalwan/ROPgadget)

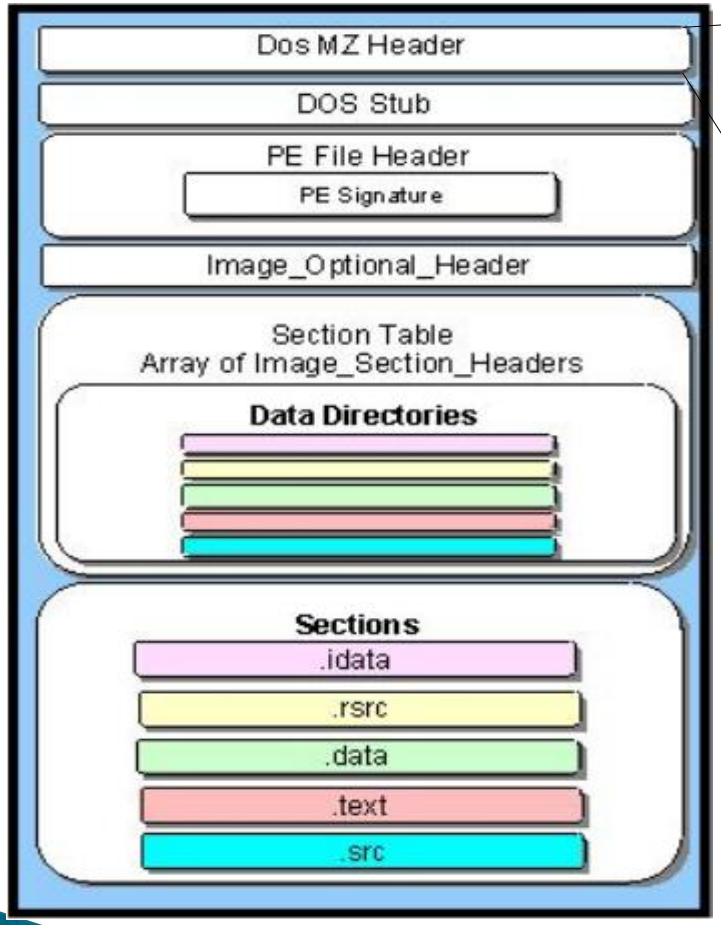
  Ropper (https://github.com/sashs/Ropper)

Exploiting Large Binaries

Binary Testing Methods

- White Box Testing.
  - Testing with full knowledge.
  - Access to source code & architecture documents.
- Black Box Testing.
  - Without knowledge of specification.
  - No access to the source code & architecture.
  - Attacker model.
- Grey Box Testing.

Windows PE Format



Header Information

Signature:	5a4d
Last Page Size:	0090
Total Pages in File:	0003
Relocation Items:	0000
Paragraphs in Header:	0004
Minimum Extra Paragraphs:	0000
Maximum Extra Paragraphs:	ffff
Initial Stack Segment:	0000
Initial Stack Pointer:	00b8
Complemented Checksum:	0000
Initial Instruction Pointer:	0000
Initial Code Segment:	0000
Relocation Table Offset:	0040
Overlay Number:	0000
Reserved:	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 8320 000c
Offset to New Header:	00000080
Memory Needed:	2K

```
00000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030 00 00 00 00 00 00 00 00 20 83 0C 00 80 00 00 00  ........ ƒ..€...
```

Windows PE Format

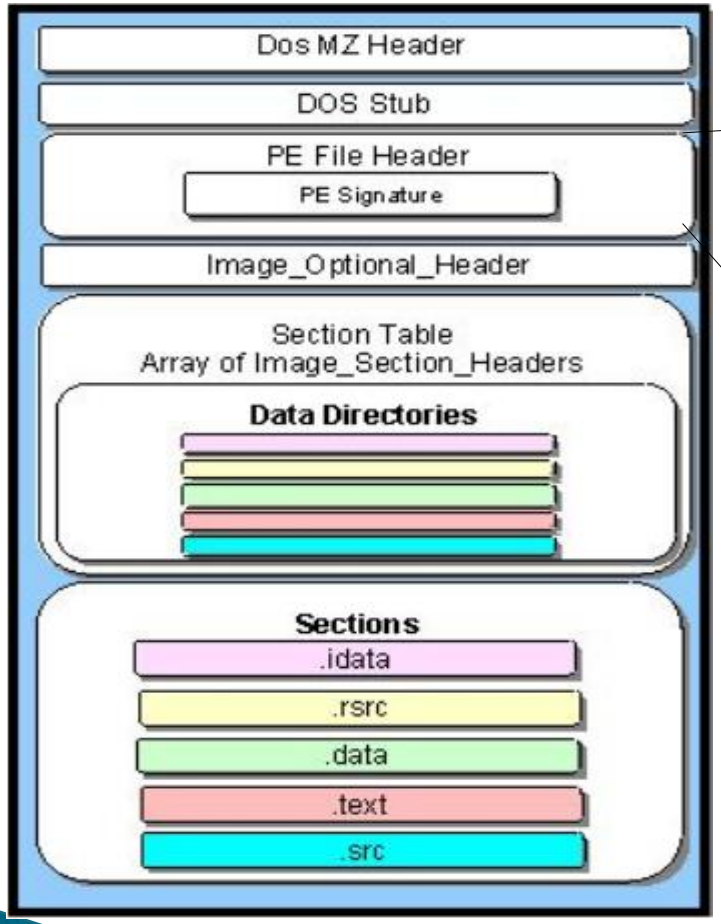


Image File Header

Signature:	00004550
Machine:	Intel 386
Number of Sections:	0004
Time Date Stamp:	00000000
Symbols Pointer:	00000000
Number of Symbols:	00000000
Size of Optional Header	00e0

00000080	50	45	00	00	4C	01	04	00	00	00	00	00	00	00	00	00	PE□□L□□□□□□□□□□□□
00000090	00	00	00	00	E0	00	0F	01	0B	01	03	00	00	02	00	00	□□□□à□□□□□□□□□□□

Windows PE Format

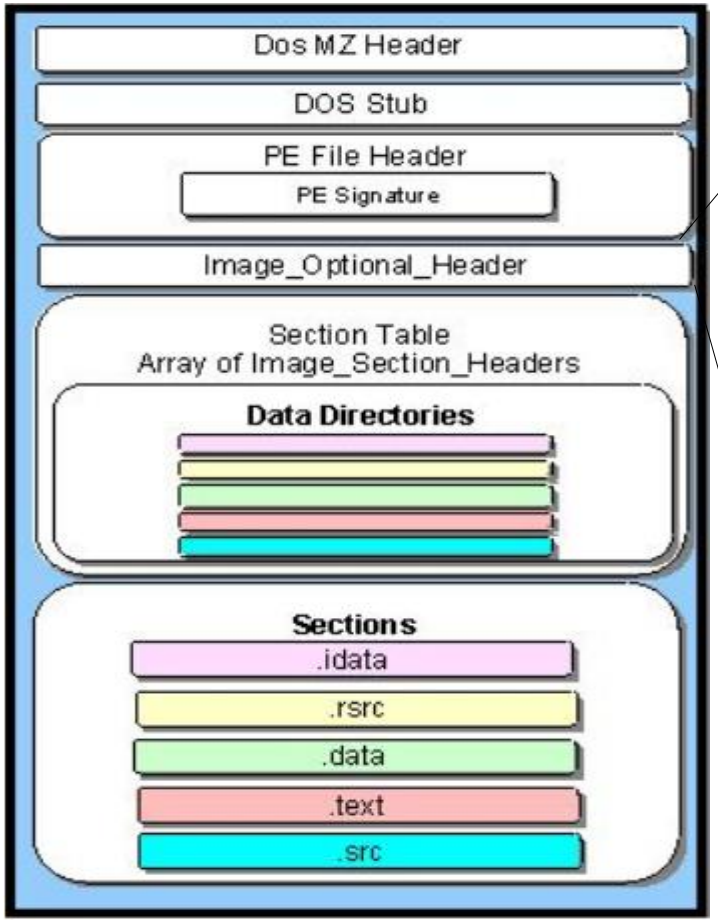
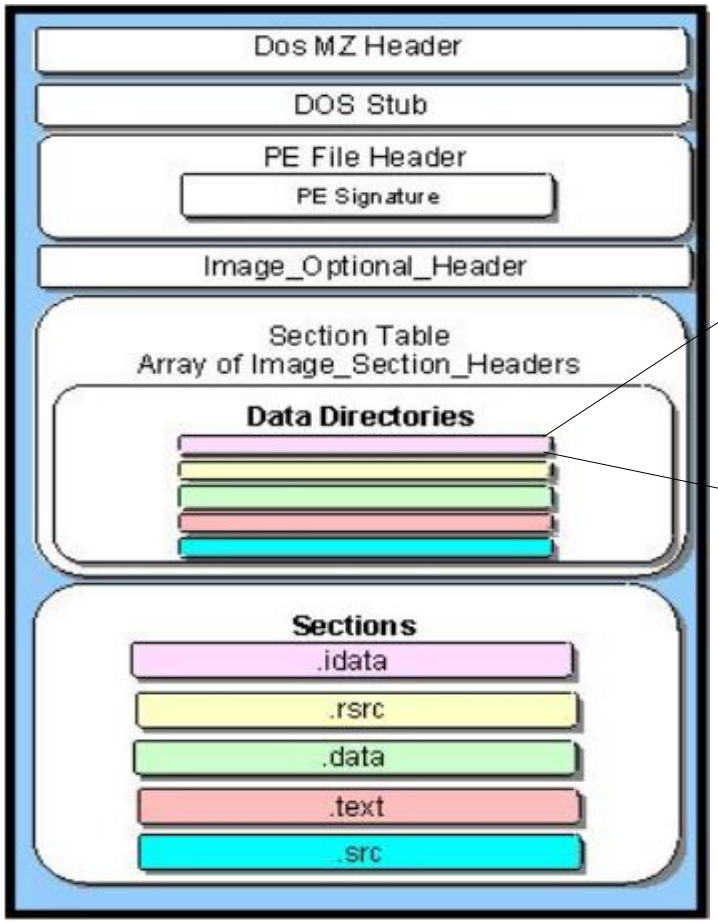


Image Optional Header

Magic:	010b
Linker Version:	3.00
Size of Code:	00000200
Size of Initialized Data:	00000e00
Size of Uninitialized Data:	00000000
Address of Entry Point:	00004000
Base of Code:	00004000
Base of Data:	00003000
Image Base:	00400000
Section Alignment:	00001000
File Alignment:	00000200
Operating System Version:	4.00
Image Version:	1.00
Subsystem Version:	4.00
Reserved1:	00000000
Size of Image:	00005000
Size of Headers:	00000400
Checksum:	00000000
Subsystem:	Image runs in the Windows GUI subsystem.
DLL Characteristics:	0000
Size of Stack Reserve:	00100000
Size of Stack Commit:	00001000
Size of Heap Reserve:	00005000
Size of Heap Commit:	00001000
Loader Flags:	00000000
Size of Data Directory:	00000010
Import Directory Virtual Address:	1000
Import Directory Size:	003c
Resource Directory Virtual Address:	2000
Resource Directory Size:	0800

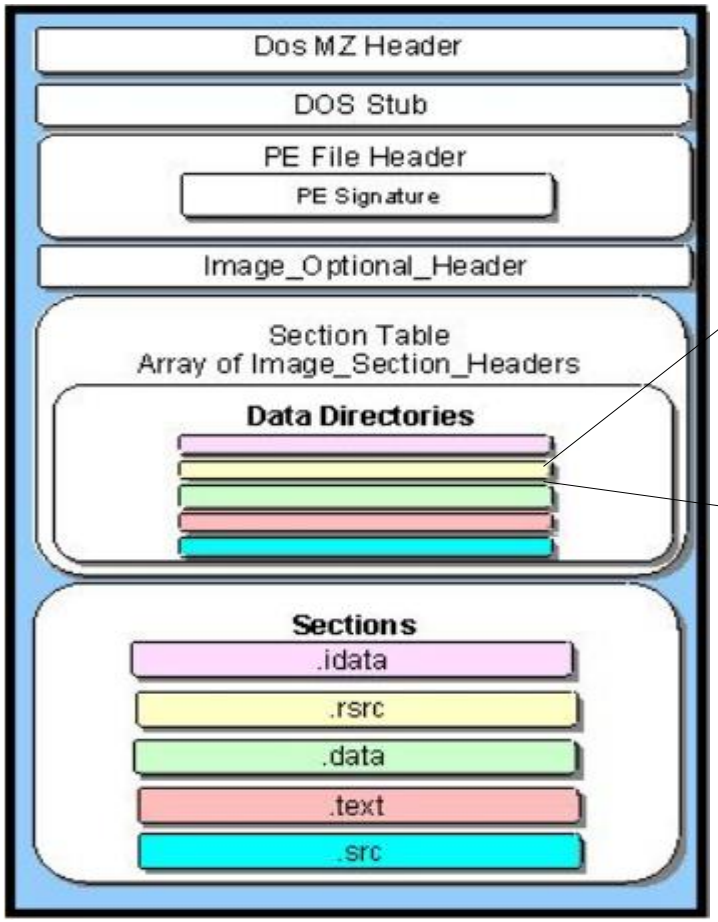
Windows PE Format



Section name:	.idata
Virtual Size:	00000204
Virtual Address:	00001000
Size of raw data:	00000400
Pointer to Raw Data:	00000400
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains initialized data Section is readable Section is writeable

```
00000170 00 00 00 00 00 00 00 00 2E 69 64 61 74 61 00 00  .........idata..
00000180 04 02 00 00 00 10 00 00 00 04 00 00 00 04 00 00  ................
00000190 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0  ............@..À
```

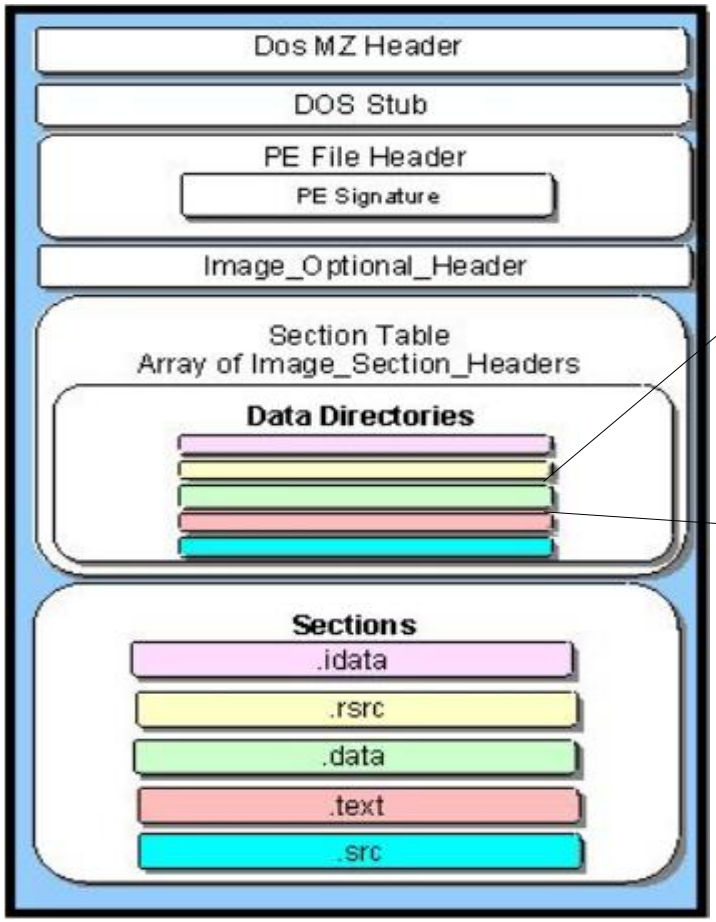
Windows PE Format



Section name:	.rsrc
Virtual Size:	00000680
Virtual Address:	00002000
Size of raw data:	00000800
Pointer to Raw Data:	00000800
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains initialized data Section is readable

```
000001A0  2E 72 73 72 63 00 00 00 80 06 00 00 00 20 00 00  .rsrc□□□□□□□□ □□
000001B0  00 08 00 00 00 08 00 00 00 00 00 00 00 00 00 00  □□□□□□□□□□□□□□□□
000001C0  00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00  □□□□@□□@.data□□□
```

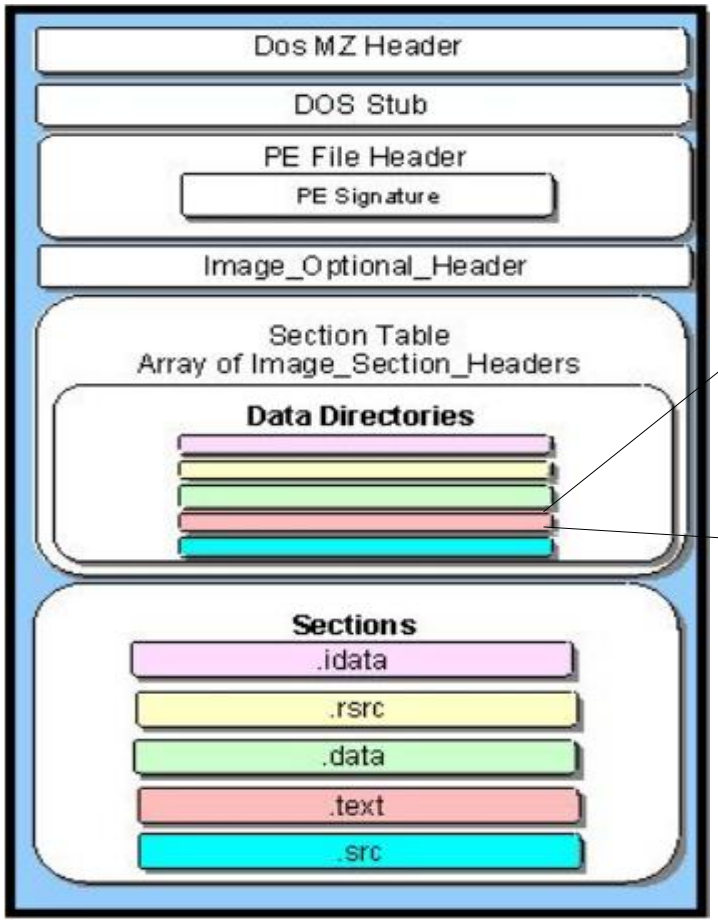
Windows PE Format



Section name:	.data
Virtual Size:	00000129
Virtual Address:	00003000
Size of raw data:	00000200
Pointer to Raw Data:	00001000
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains initialized data Section is readable Section is writeable

```
000001C0 00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00 ....@..@.data...
000001D0 29 01 00 00 00 30 00 00 00 02 00 00 00 10 00 00 )....0..........
000001E0 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0 ............@..À
```

Windows PE Format



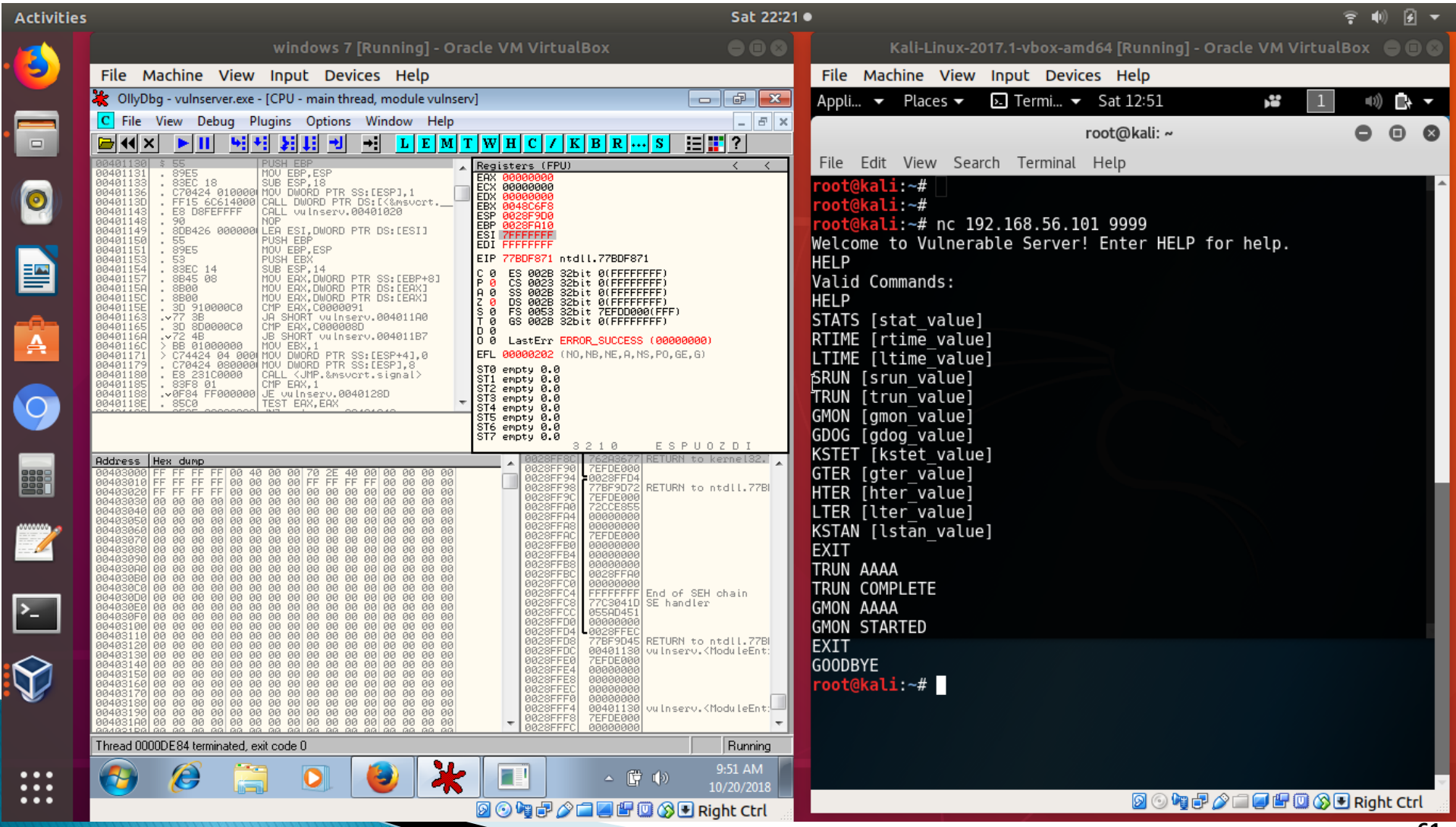
Section name:	.text
Virtual Size:	000001ec
Virtual Address:	00004000
Size of raw data:	00000200
Pointer to Raw Data:	00001200
Pointer to Relocations:	00000000
Pointer to Line Numbers:	00000000
Number of Relocations:	0000
Number of Line Numbers:	0000
Characteristics:	Section contains code Section is executable Section is readable

000001F0	2E	74	65	78	74	00	00	00	EC	01	00	00	00	40	00	00	.text□□□ì□□□□@□□
00000200	00	02	00	00	00	12	00	00	00	00	00	00	00	00	00	00	□□□□□□□□□□□□□□□□
00000210	00	00	00	00	20	00	00	60	00	00	00	00	00	00	00	00	□□□□ □□`□□□□□□□□

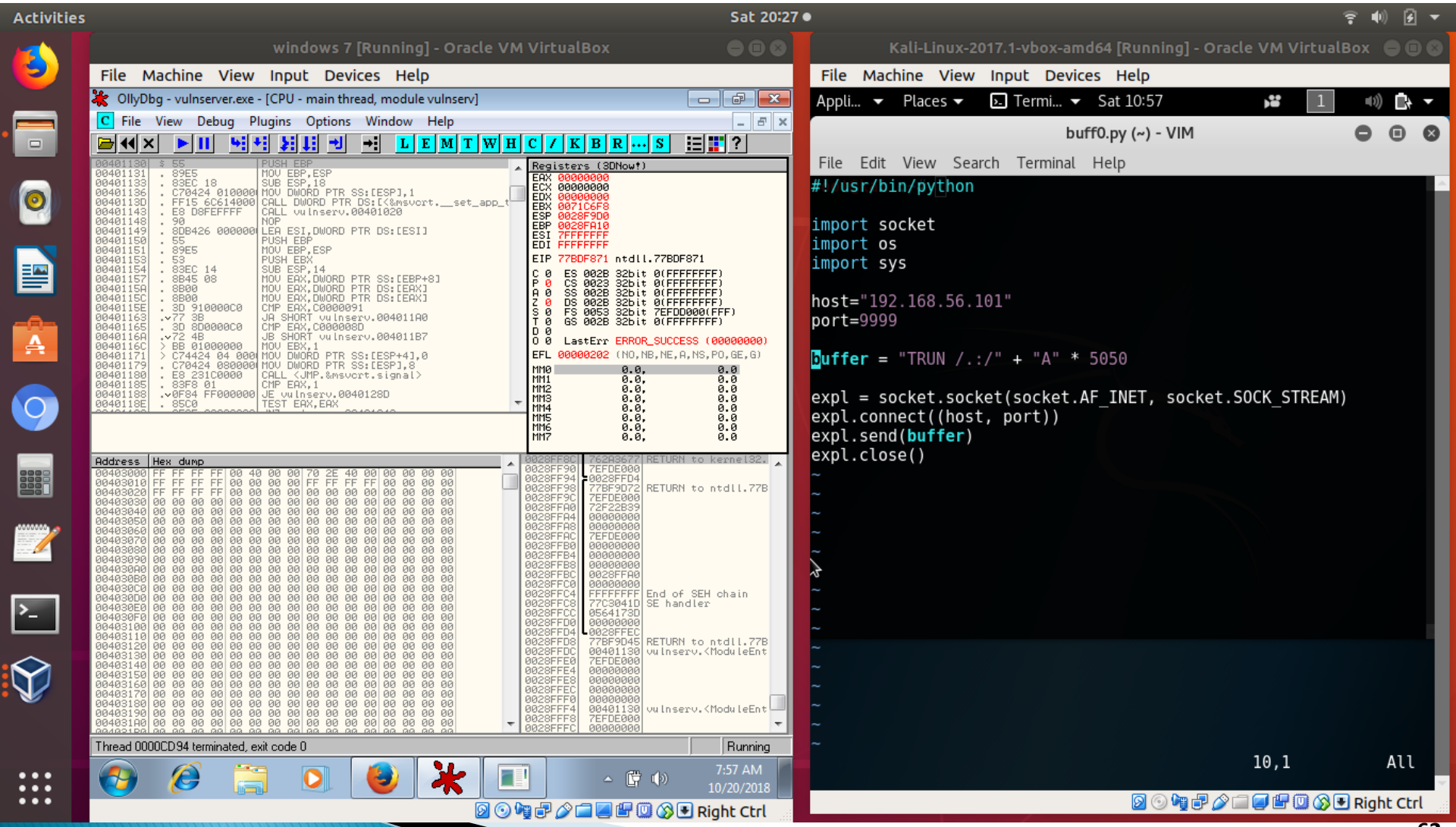
Black Box Exploitation

- Establishing a Working Environment.
- Fuzzing.
  - Input Generation.
  - Fault Injection.
  - Fault Delivery.
  - Fault Monitoring.
- Binary Auditing.

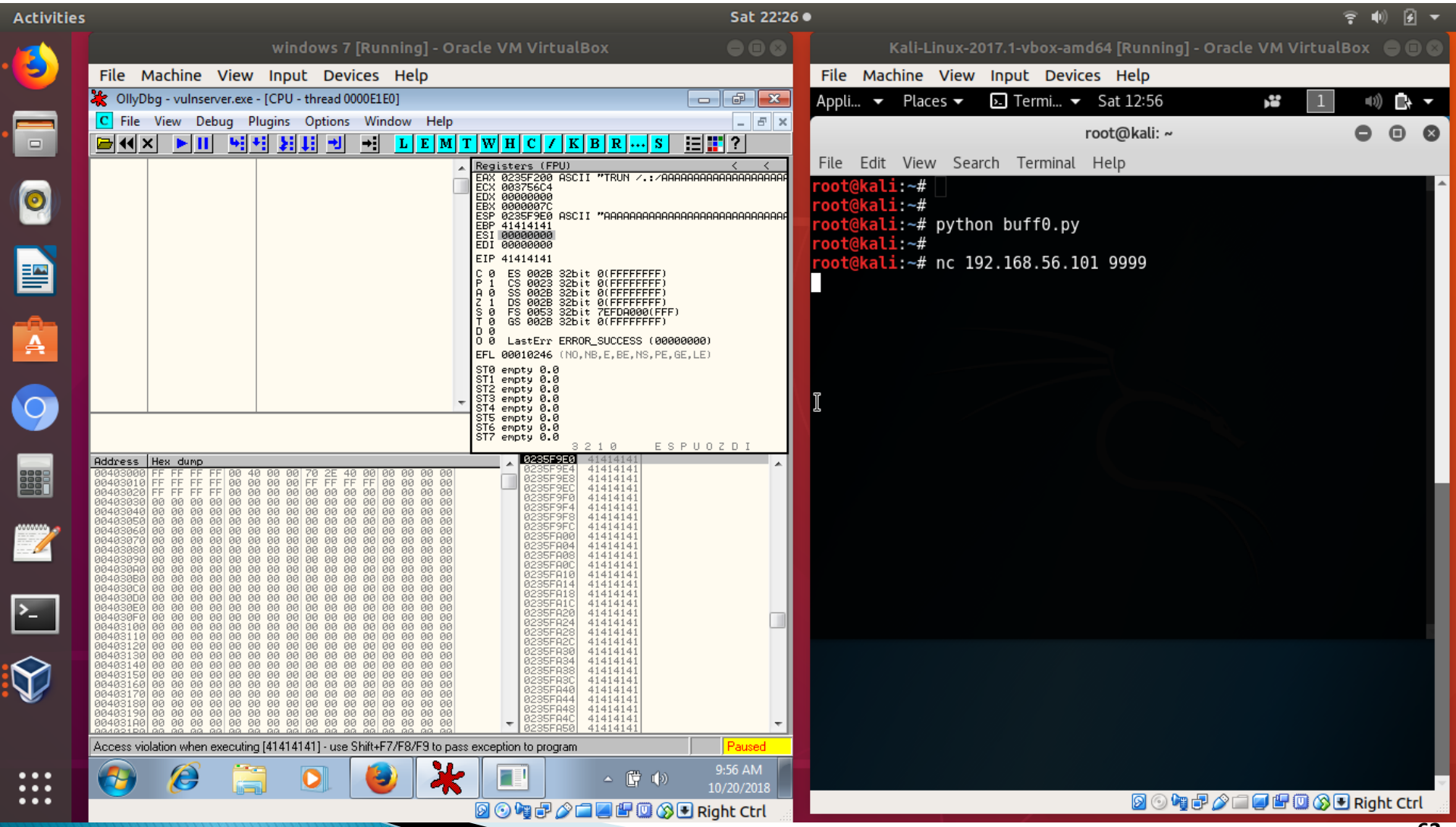
Black Box Exploitation Example



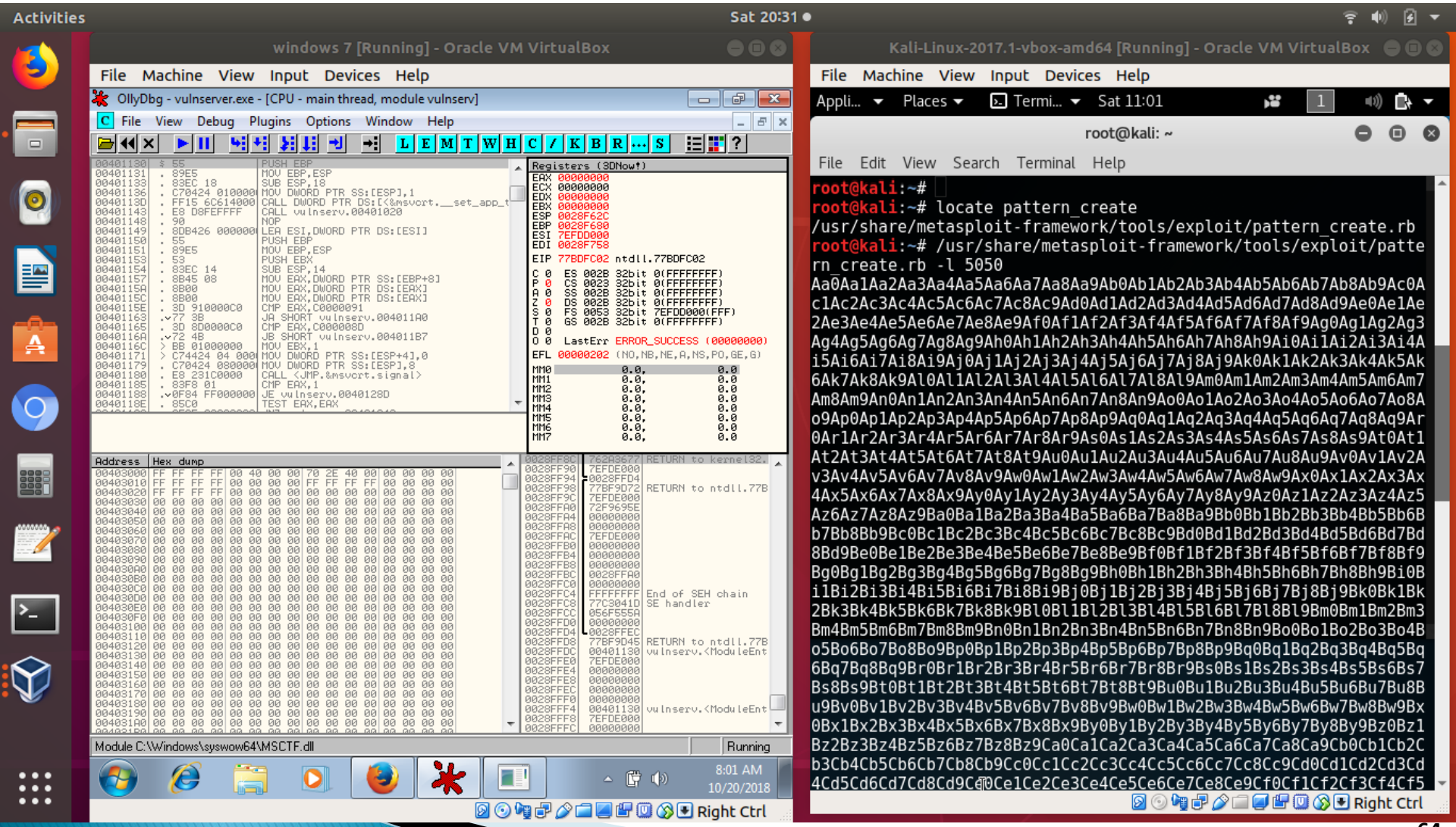
Black Box Exploitation Example



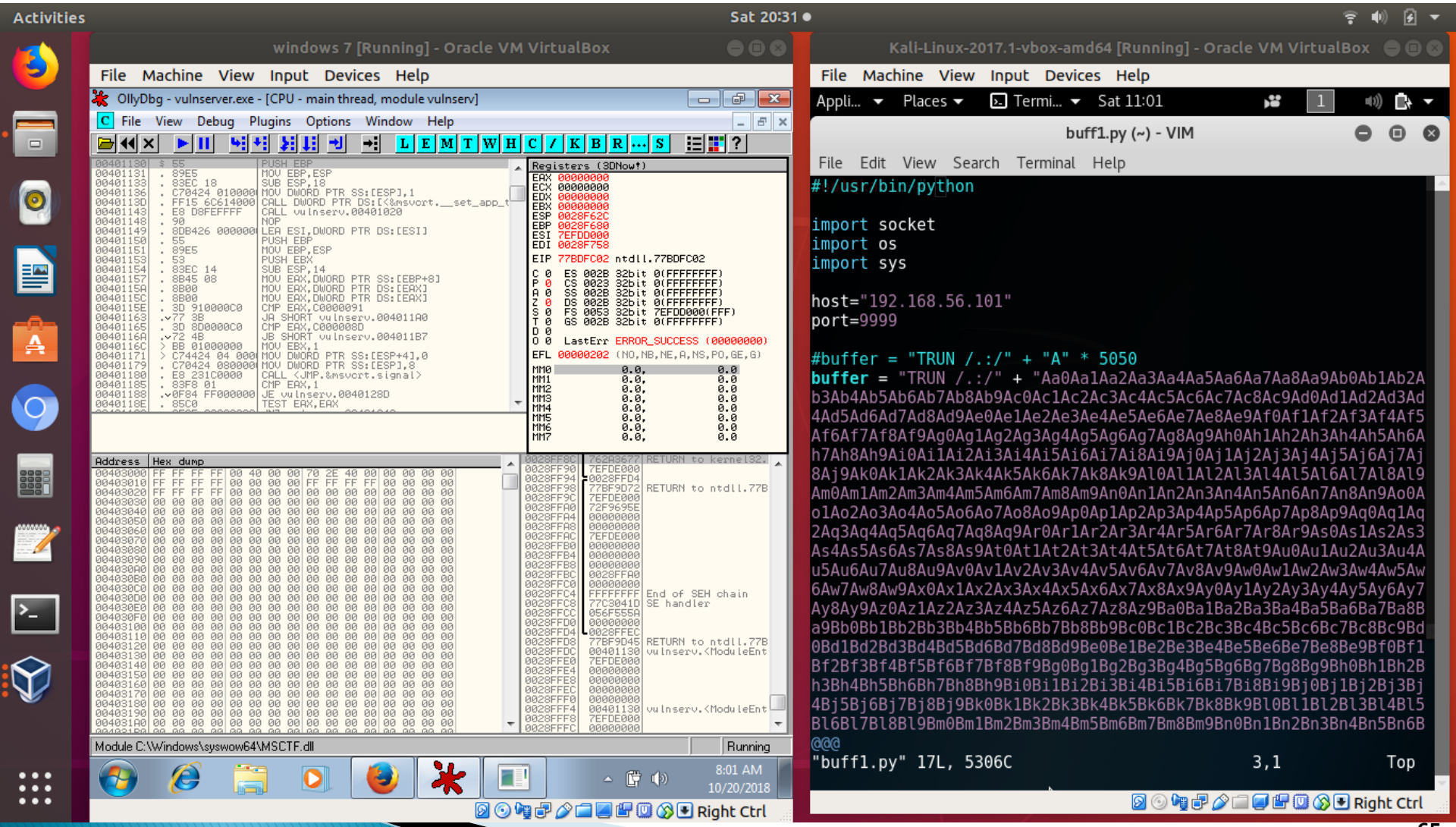
Black Box Exploitation Example



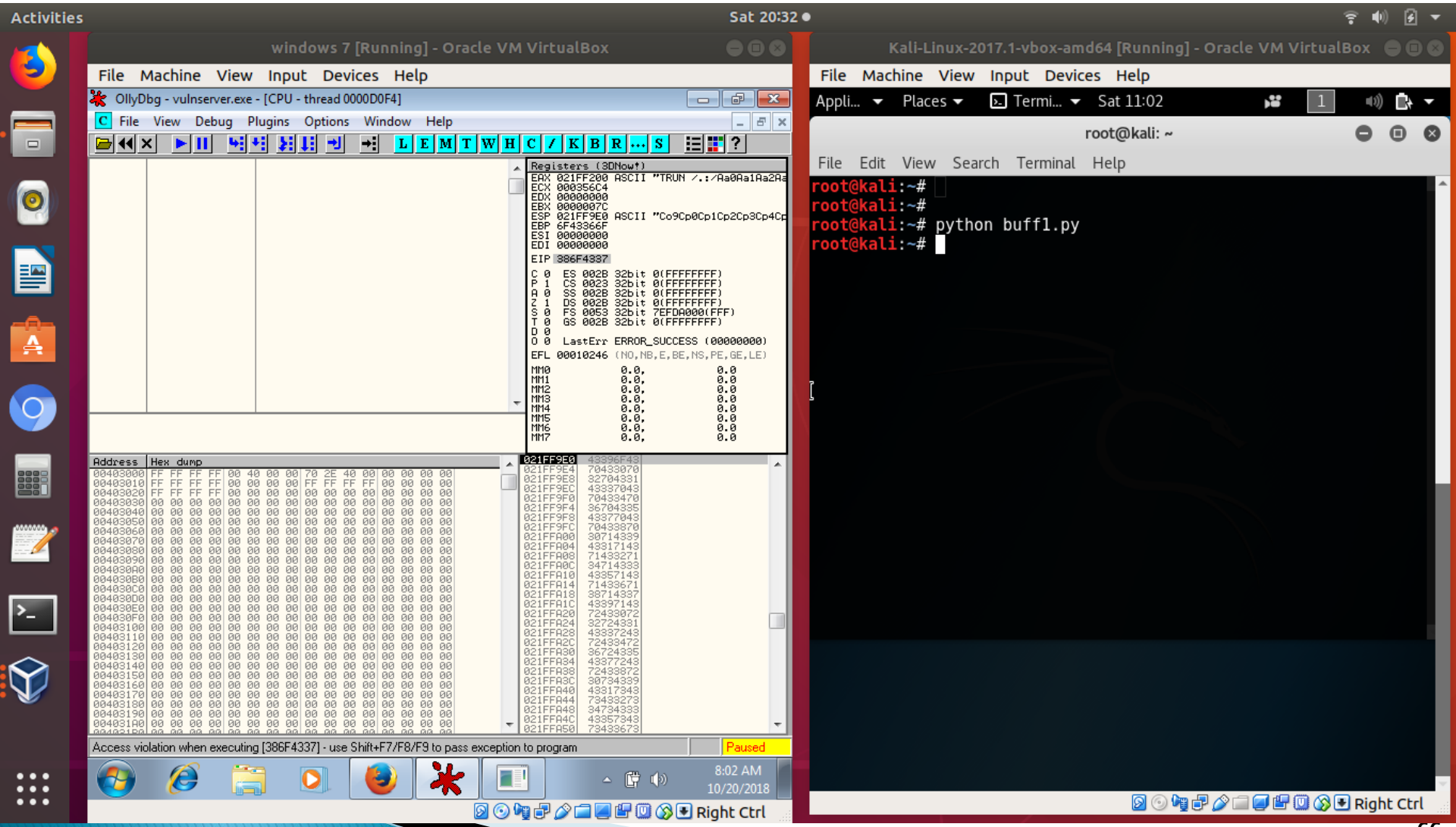
Black Box Exploitation Example



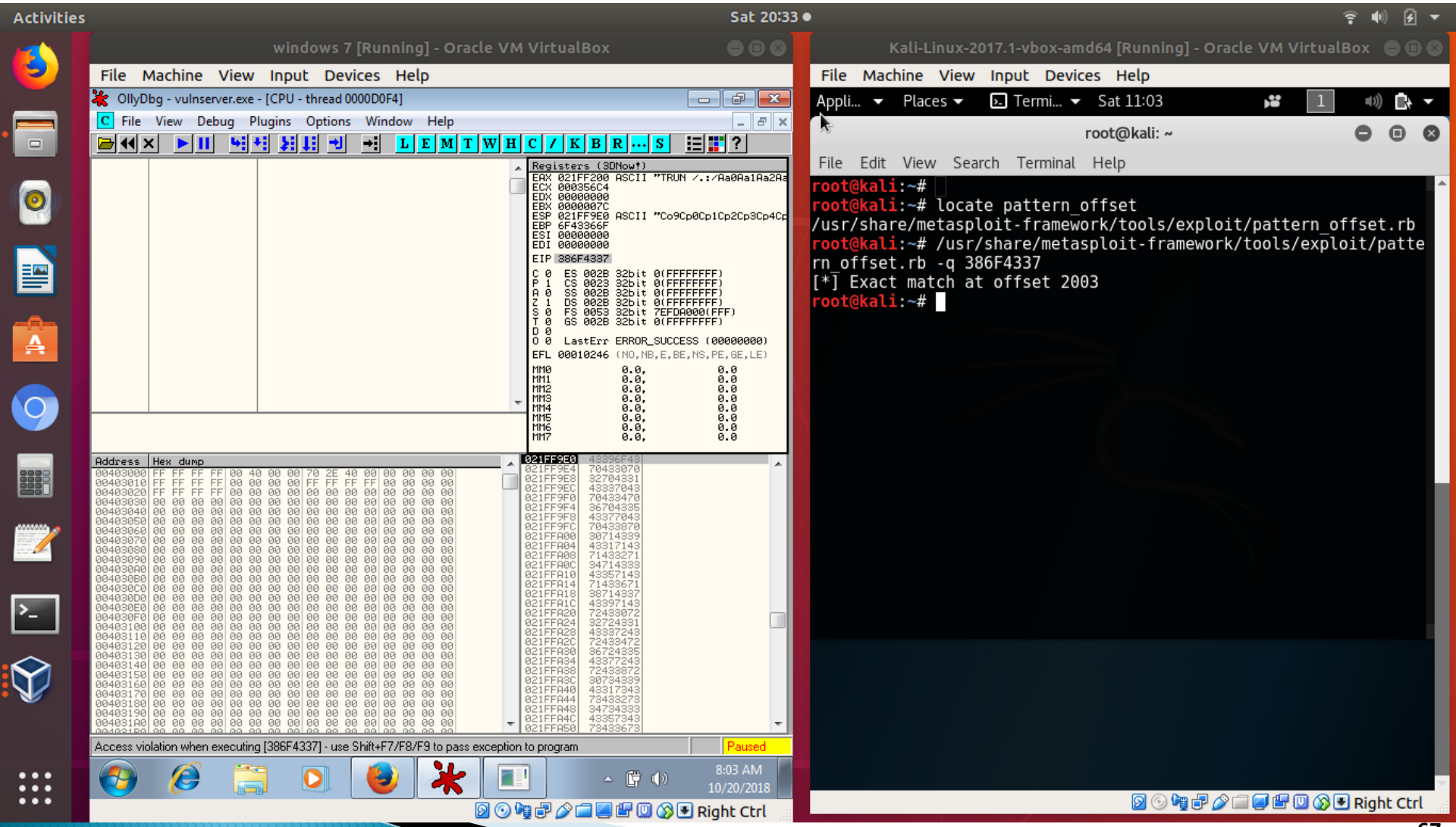
Black Box Exploitation Example



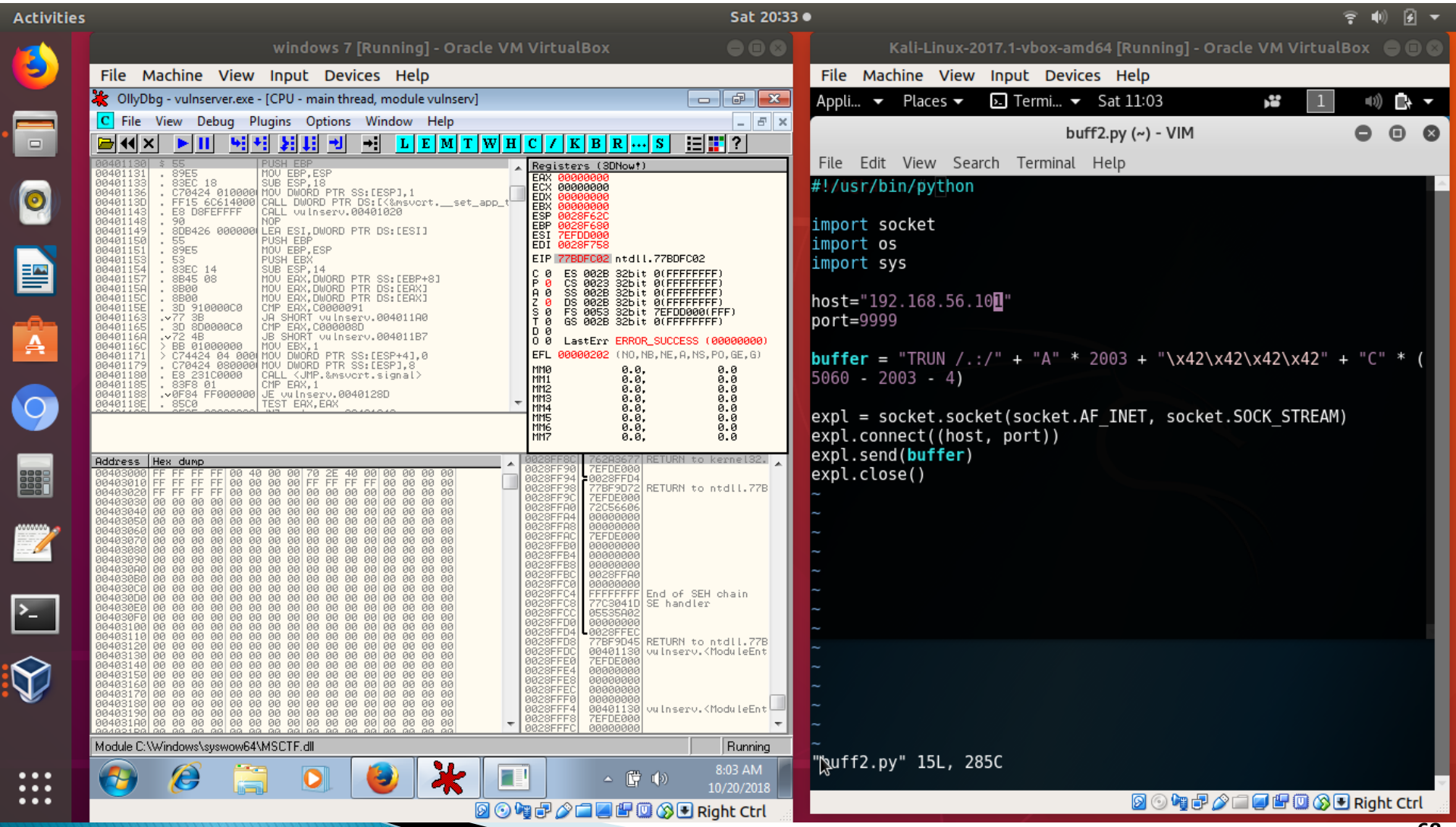
Black Box Exploitation Example



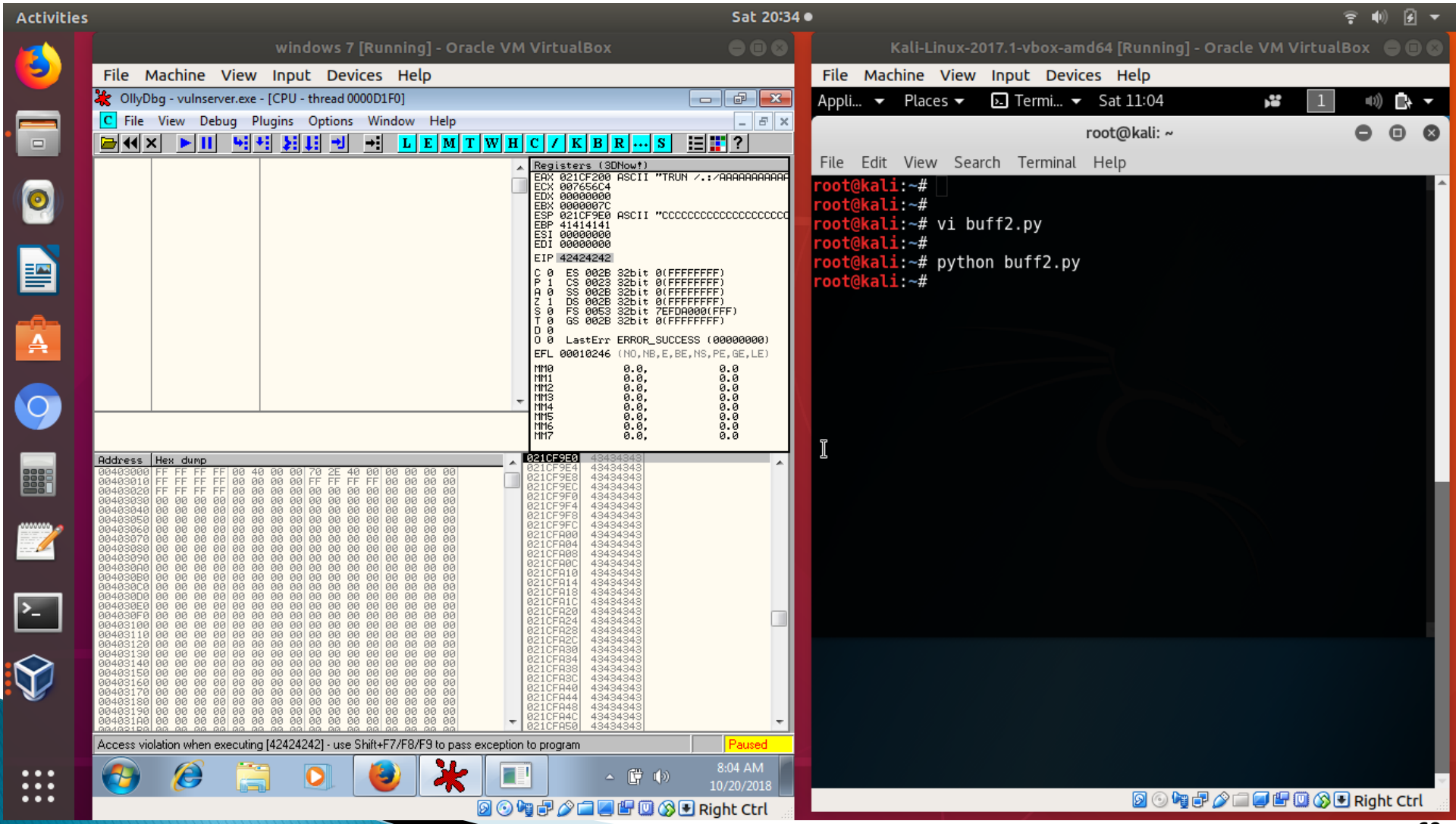
Black Box Exploitation Example



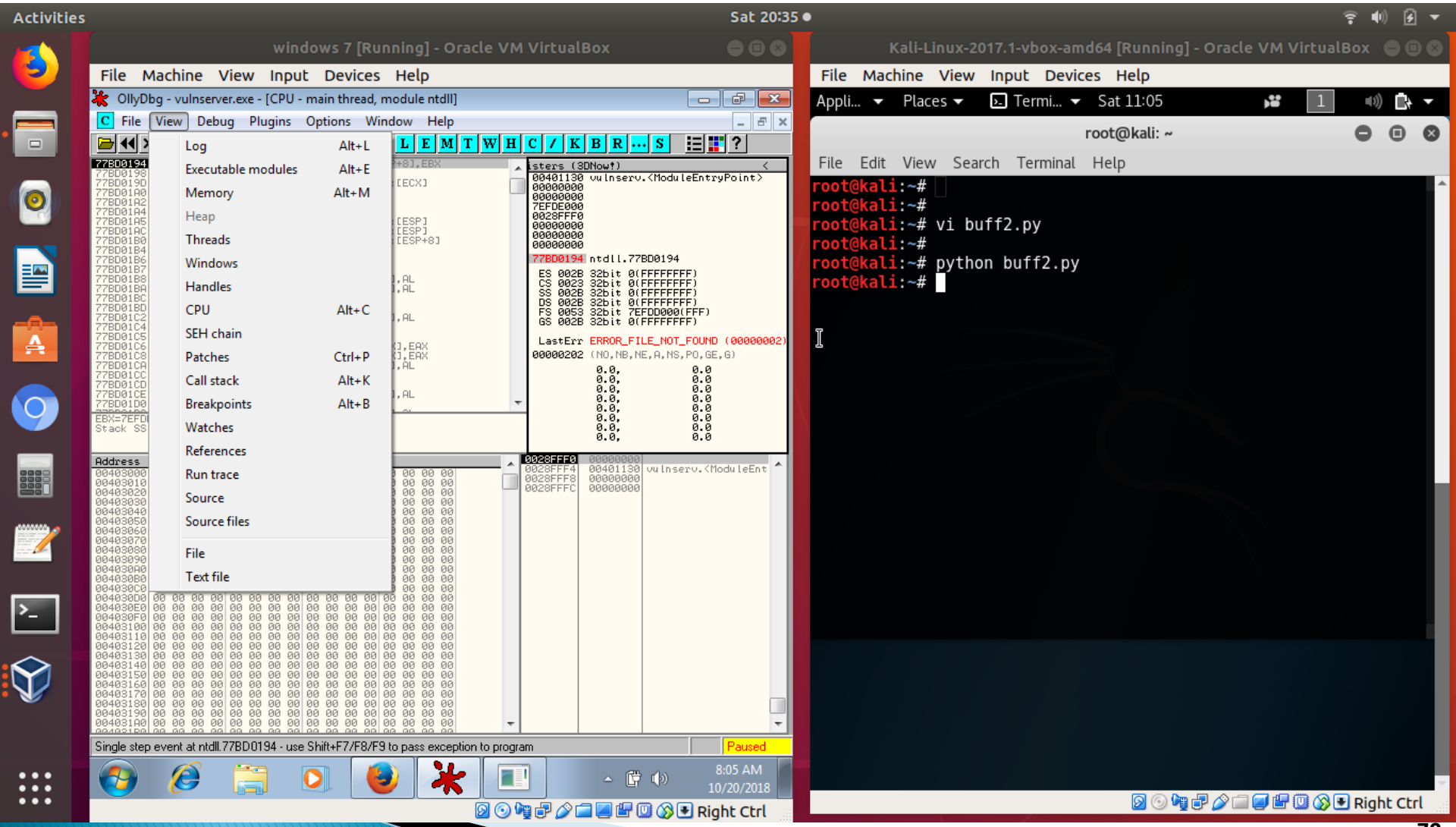
Black Box Exploitation Example



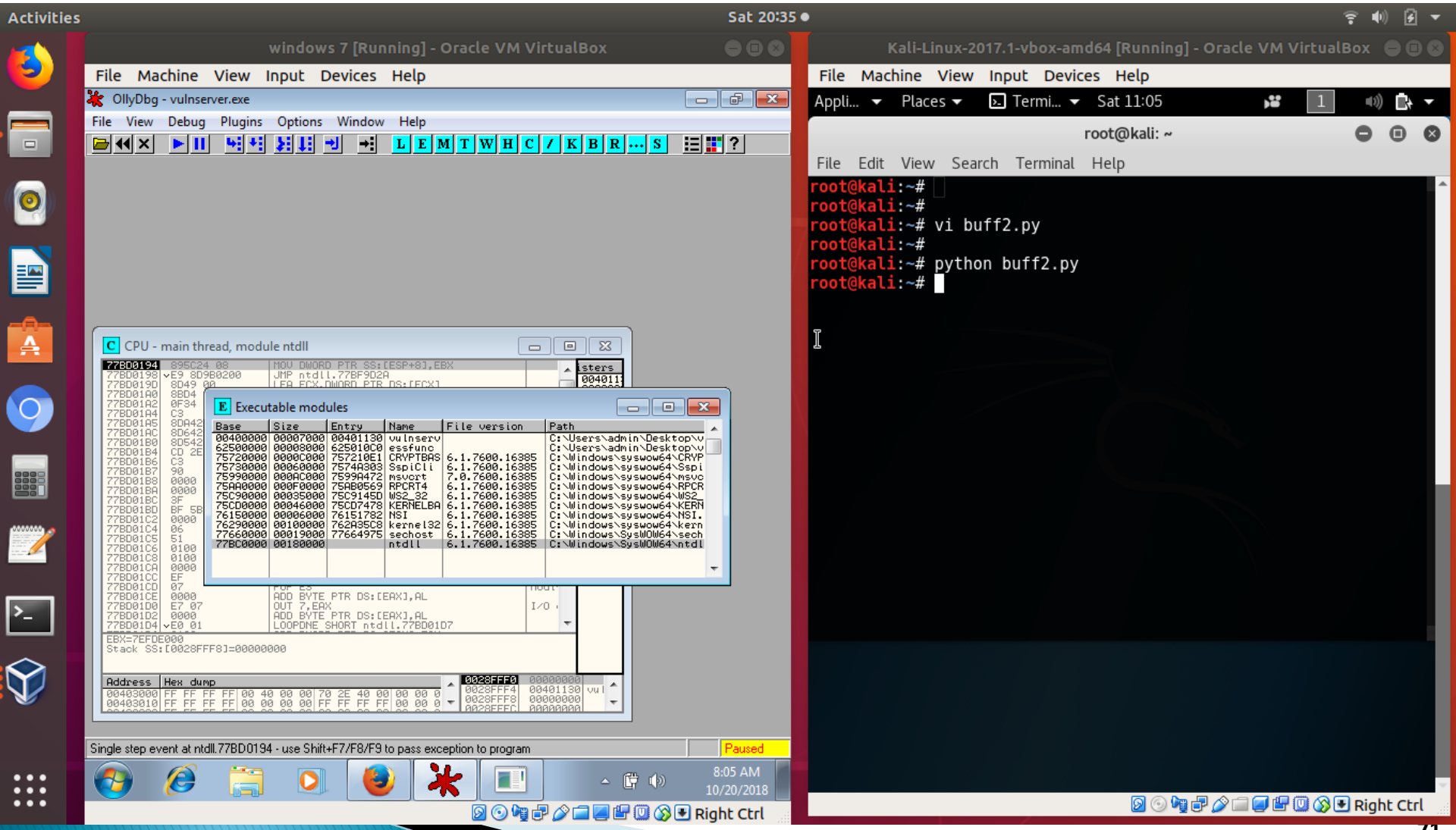
Black Box Exploitation Example



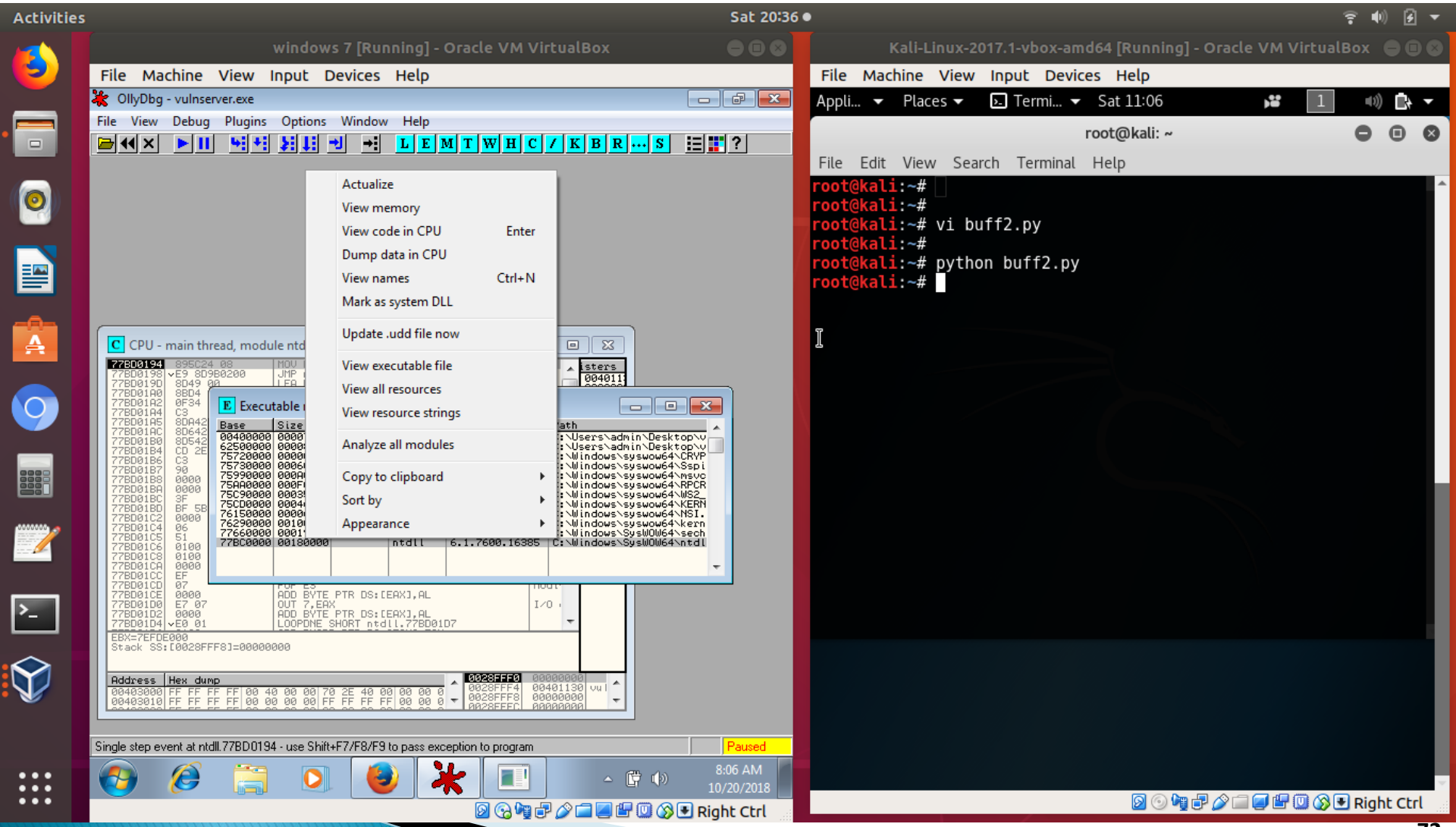
Black Box Exploitation Example



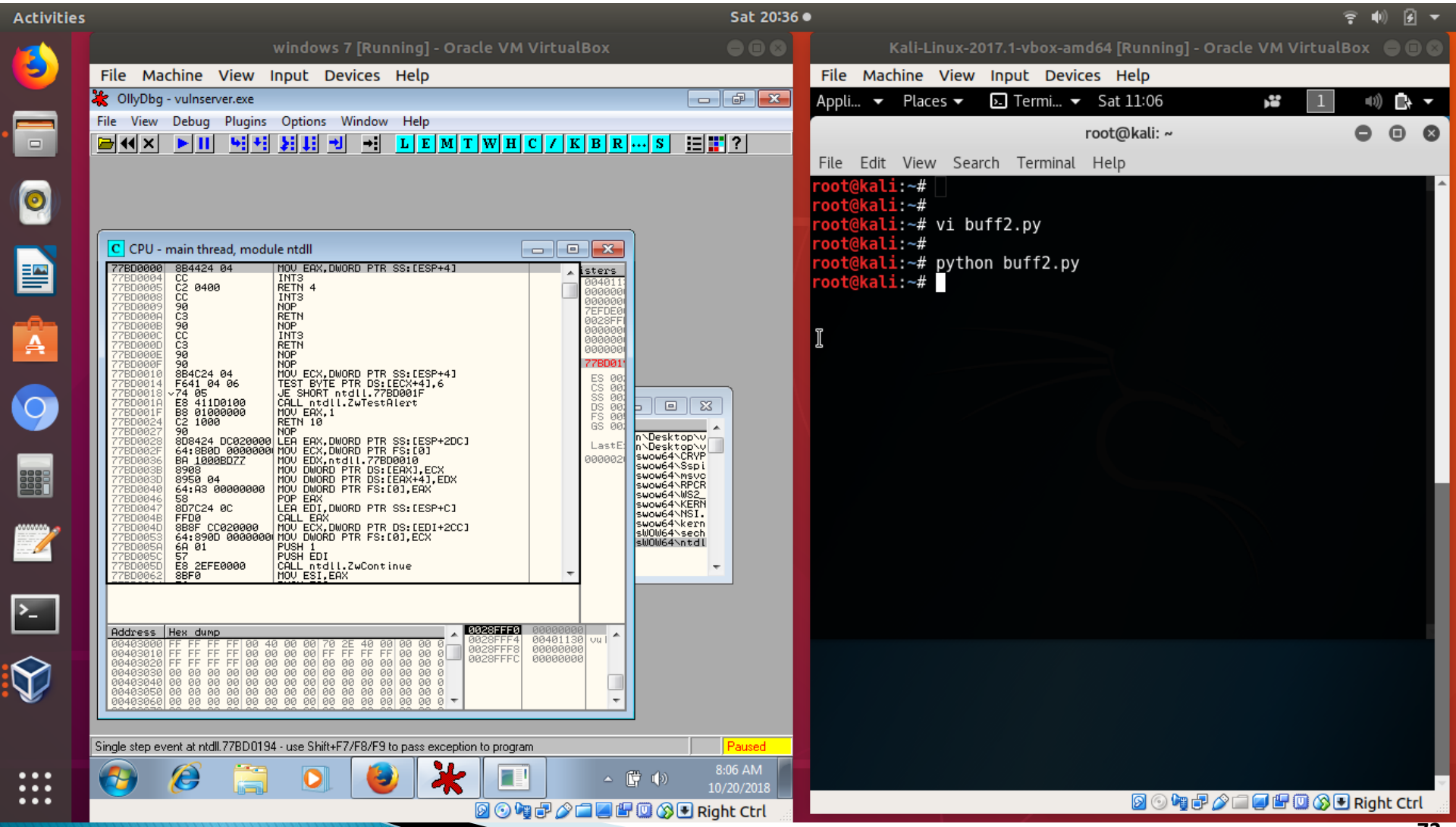
Black Box Exploitation Example



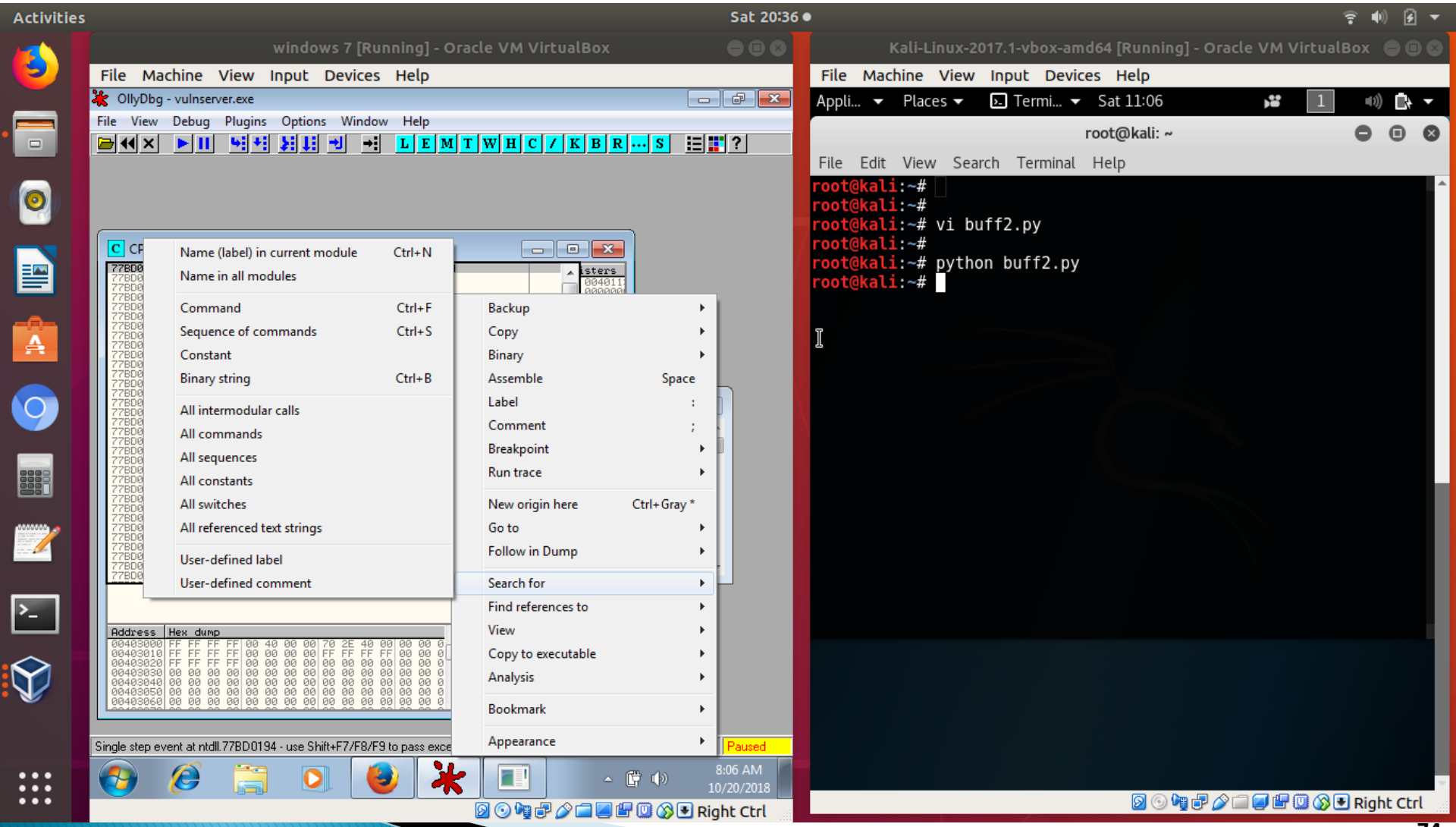
Black Box Exploitation Example



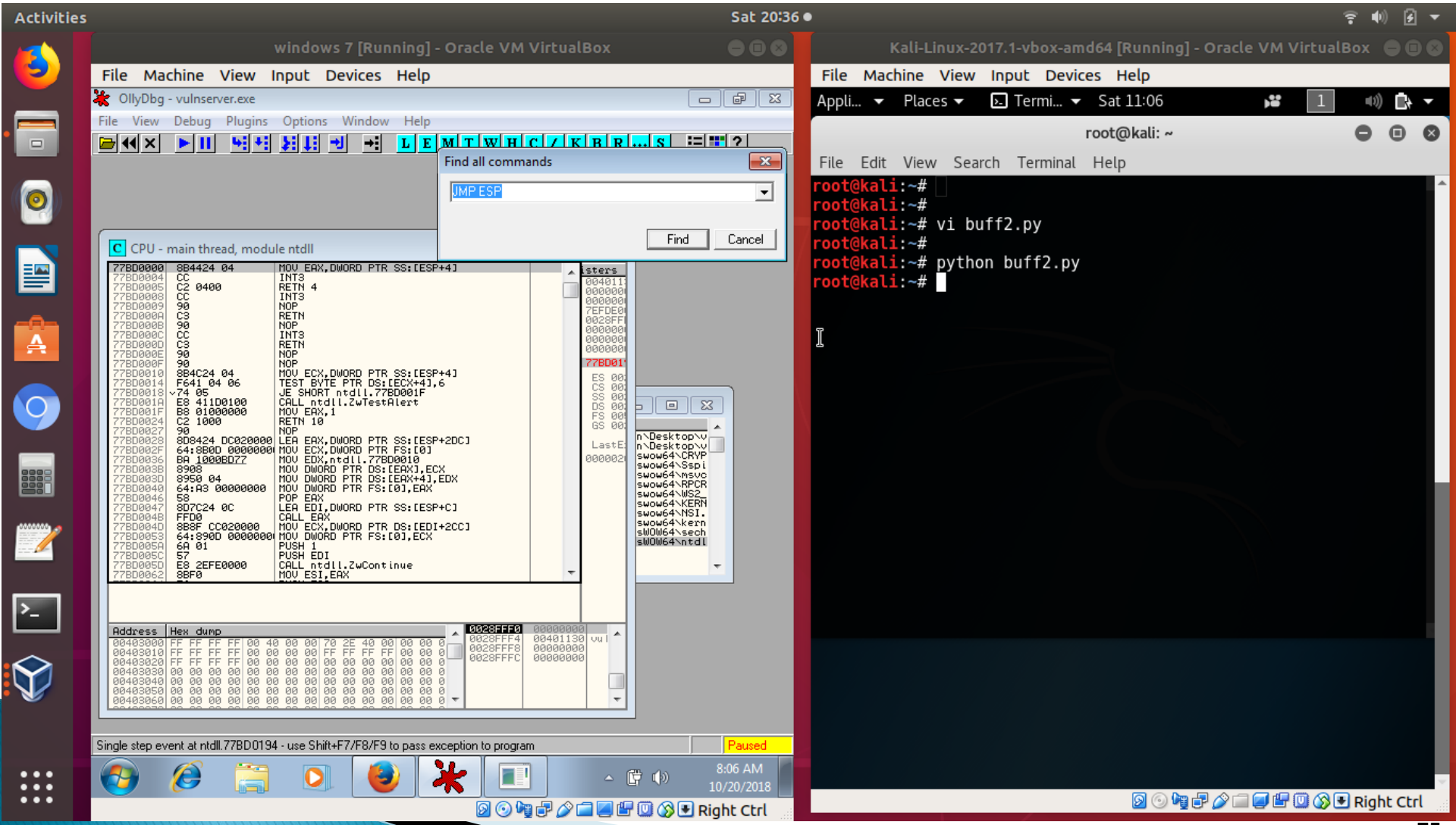
Black Box Exploitation Example



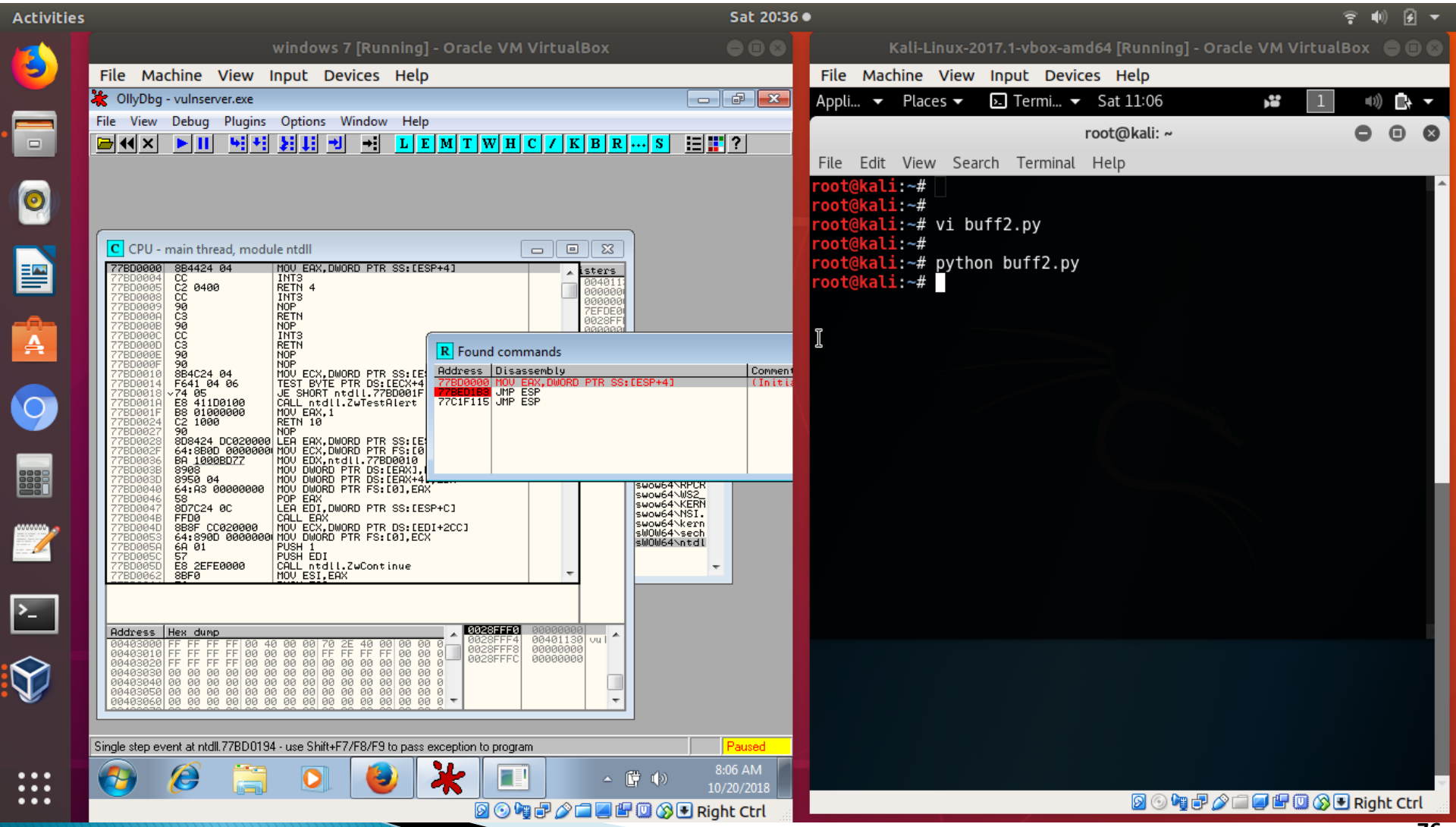
Black Box Exploitation Example



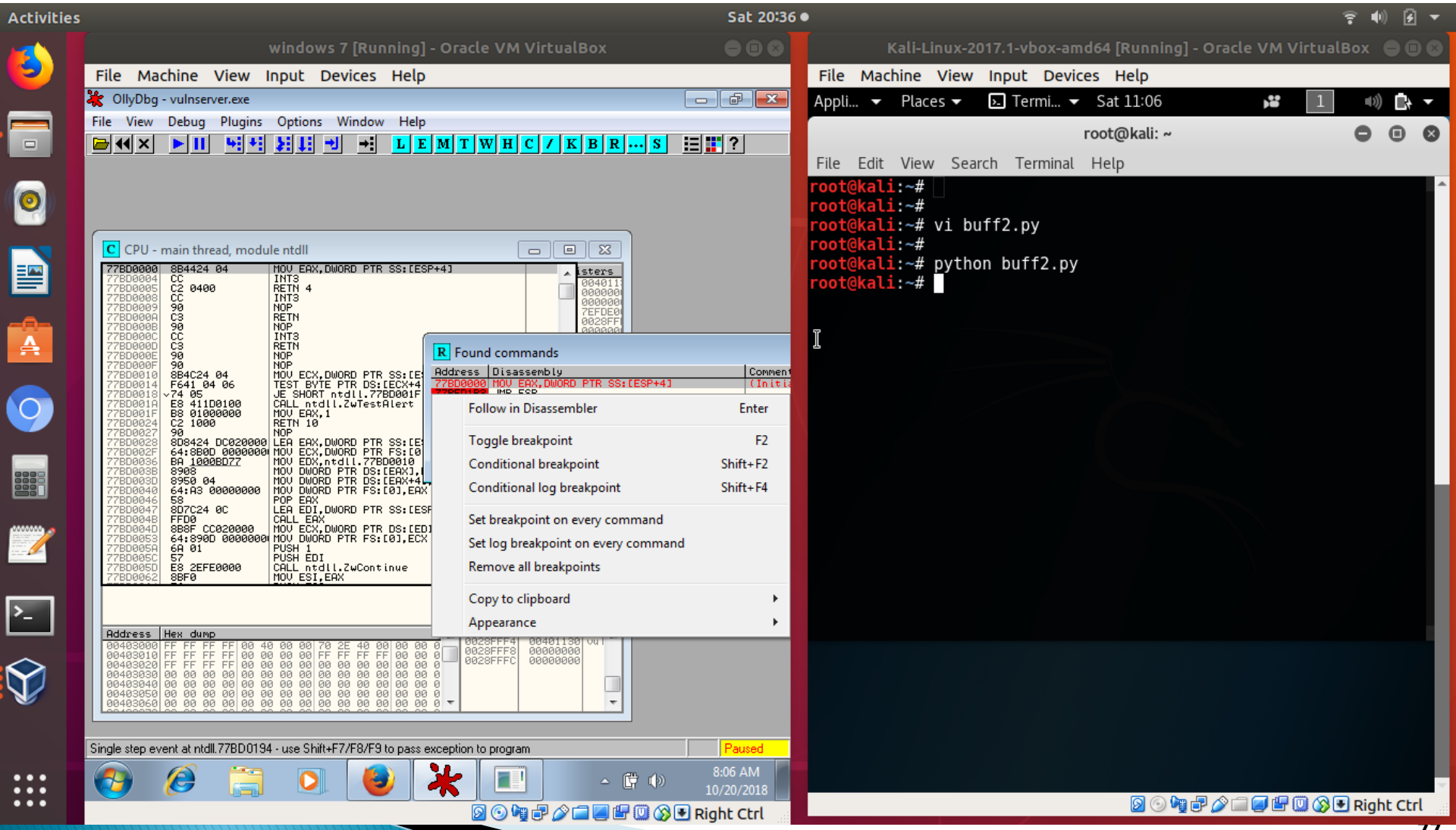
Black Box Exploitation Example



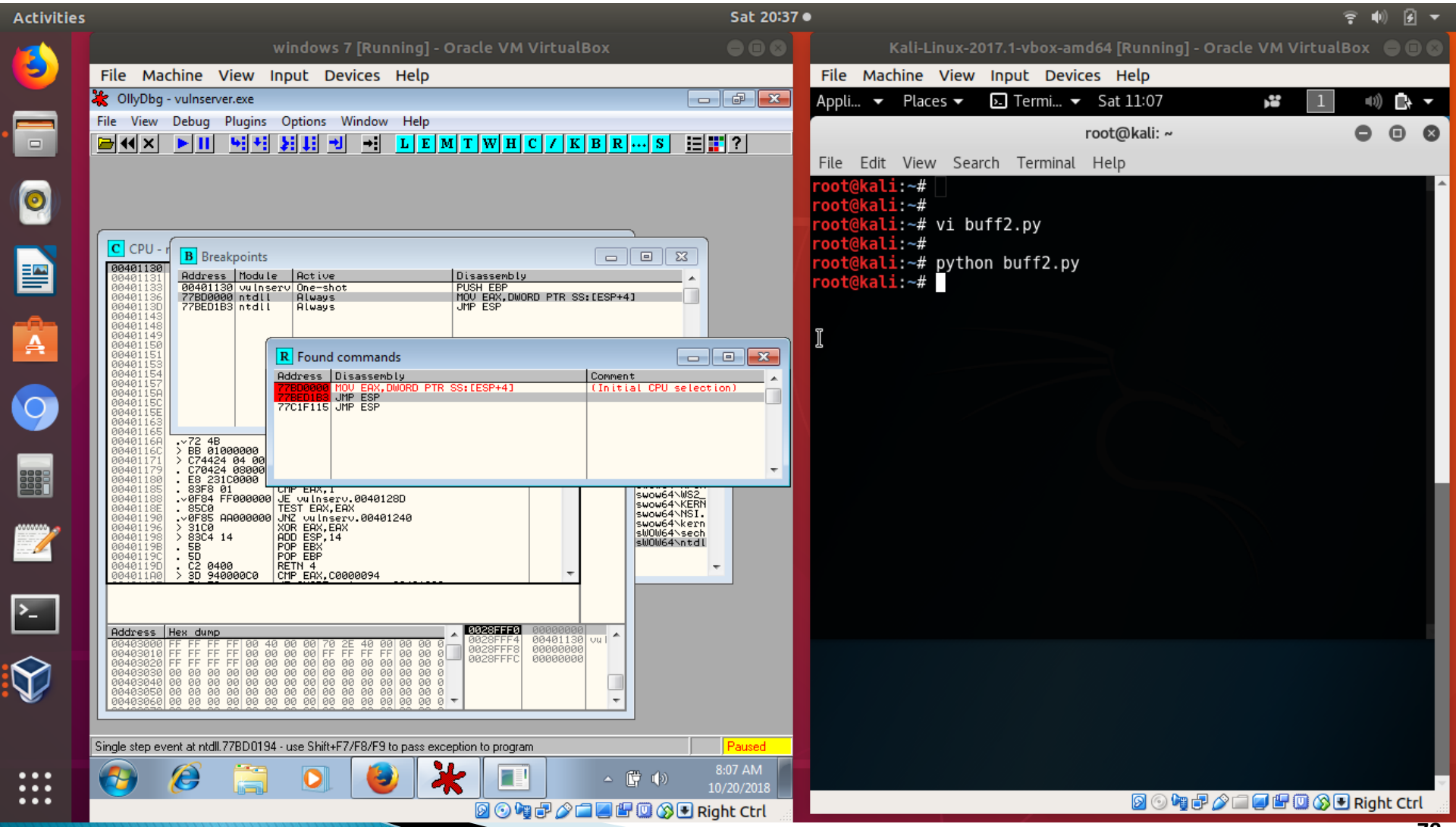
Black Box Exploitation Example



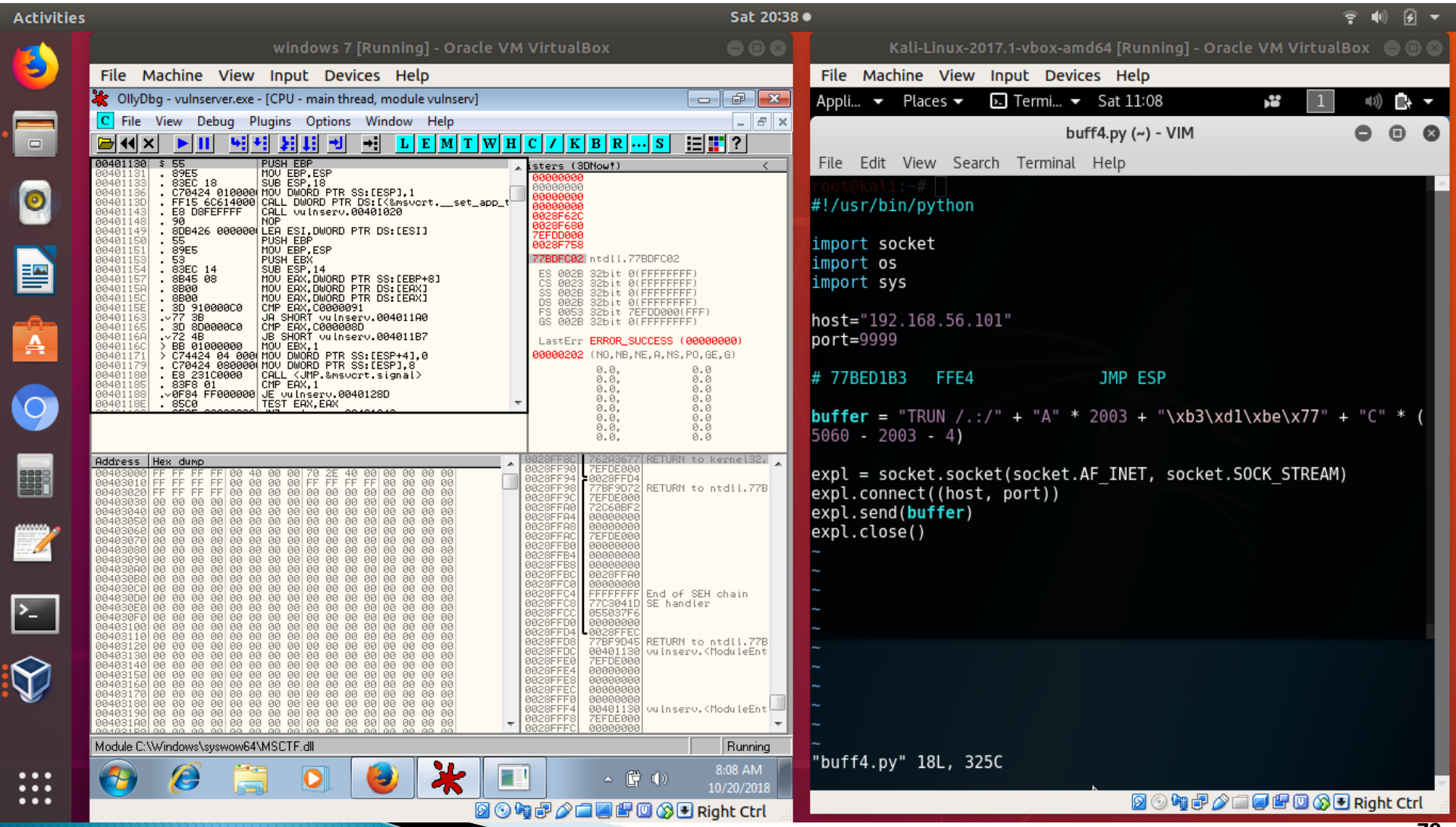
Black Box Exploitation Example



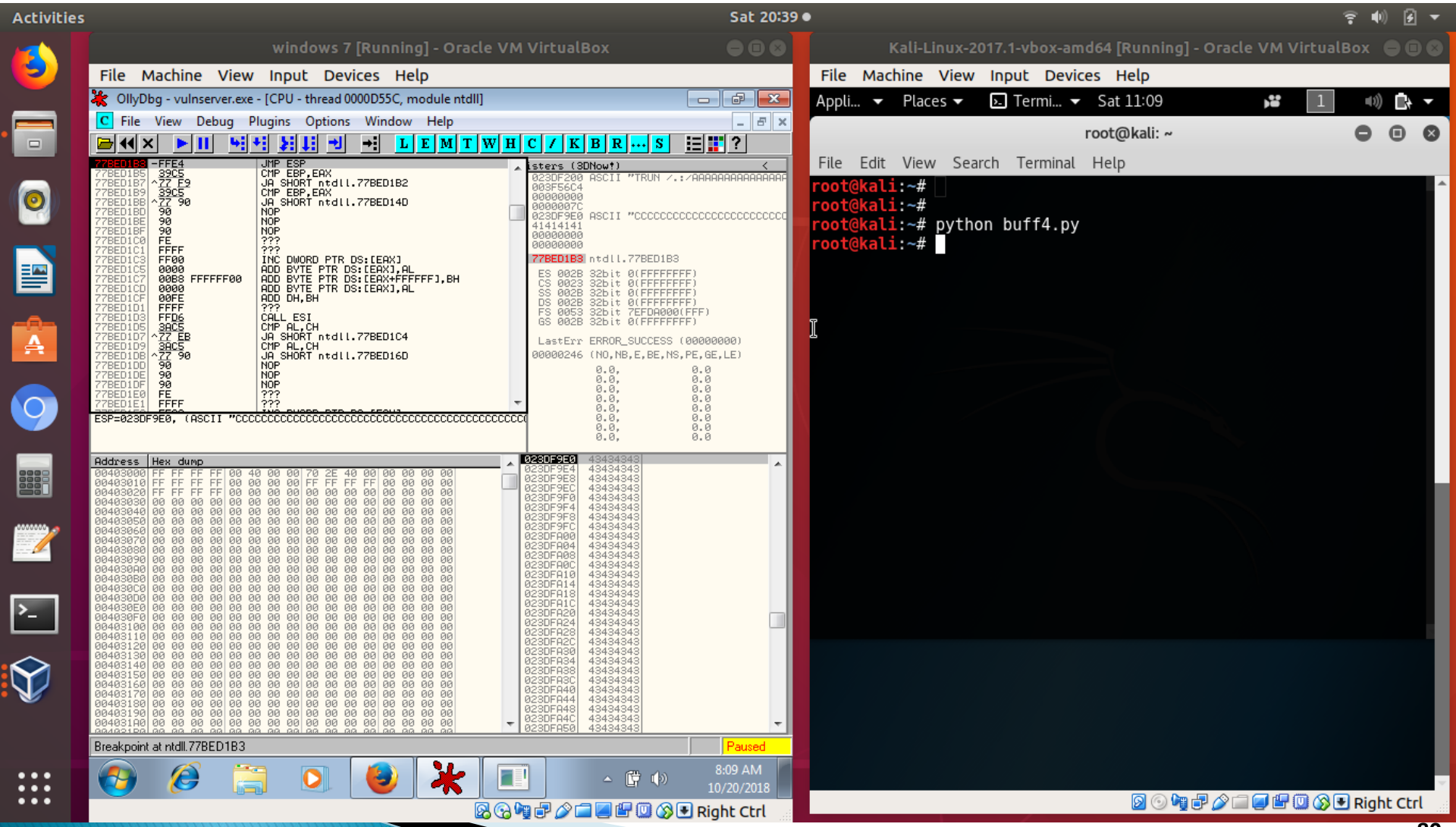
Black Box Exploitation Example



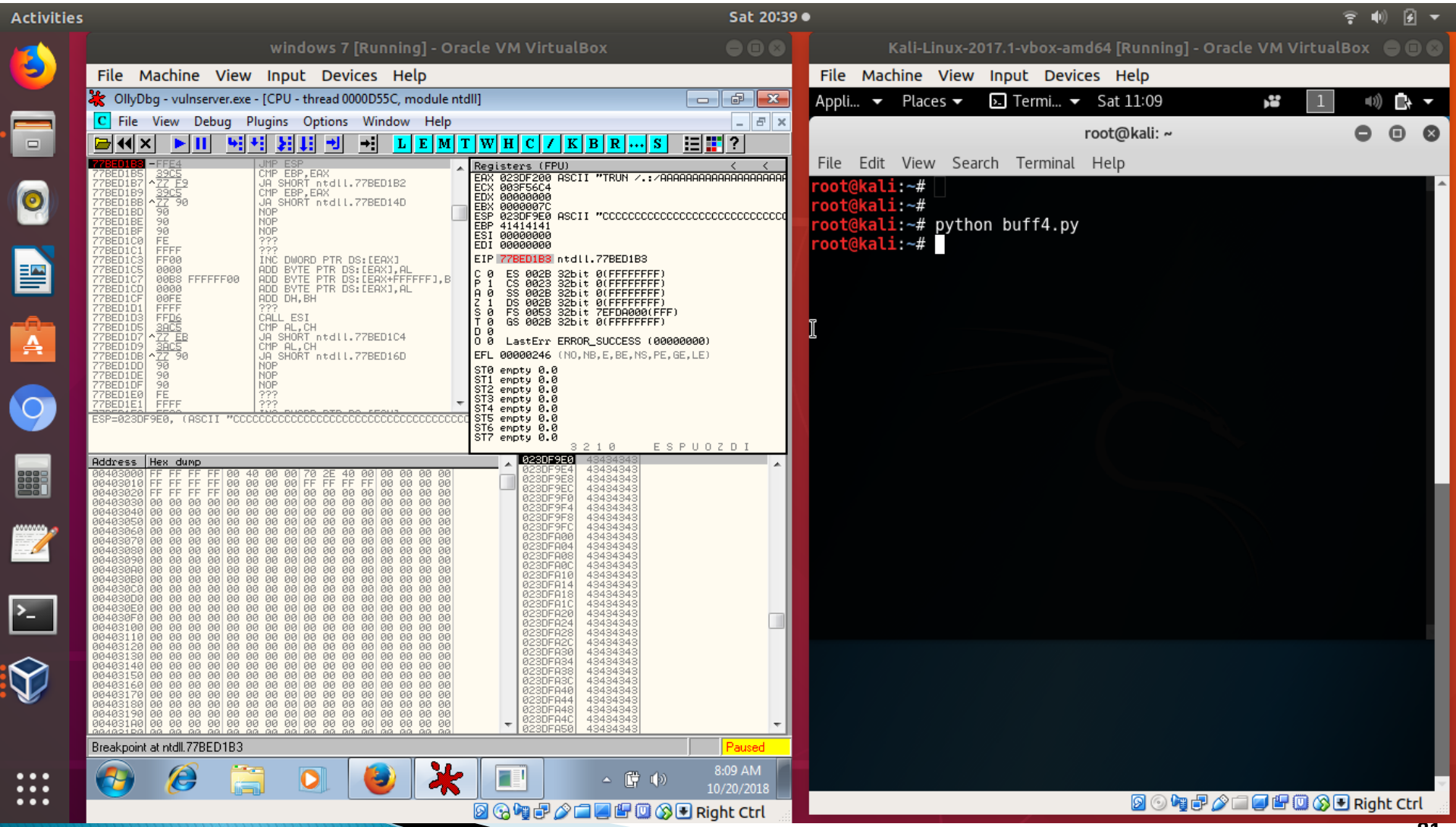
Black Box Exploitation Example



Black Box Exploitation Example



Black Box Exploitation Example



Jumping to Shellcode

- jump (or call) a register that points to the shellcode.

1. `JMP`/`CALL` register

Conditions:

- A register points to shellcode.

Example:

- `[Register]` → `[Shellcode]`
- `EIP` → `JMP/CALL [register]`
- `EIP` now points to `[register]` where shellcode.

Pros:

- Easy to apply.
- Common instruction.

Cons:

- None.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- pop return.

2. `POP RET` / `POP POP RET` / `POP POP POP RET`

Condition:

- Address at `[ESP+4]`, `[ESP+8]`, `[ESP+12]` (and so on) points to address to shellcode **OR** directly to shellcode.

Example 1:

- `[ESP]` → `[4 bytes][Address to shellcode]`.
- `EIP` → `POP [register]` followed by `RET`.
  - `POP [register]`: `ESP` now points to `old_ESP + 4`
  - `RET`: `EIP` now contains address to shellcode.
- `EIP` now points to shellcode.

Example 2:

- `[ESP]` → `[8 bytes][Address to JMP ESP][shellcode]`.
- `EIP` → `POP [register]` followed by `POP [register]` followed by `RET`:
  - `POP [register]`, `POP [register]` will get rid of 8 bytes, `new ESP` → `old ESP + 8` (Address to `JMP ESP`)
  - `RET` places address to `JMP ESP` in `EIP` and now `ESP` now points to shellcode.
  - `JMP ESP`: `EIP` now contains `ESP`
- `EIP` points to current ESP value which points to start of shellcode.

Pros:

- Straightforward (slightly harder than method 1).
- Lots of possible combinations to find instruction.

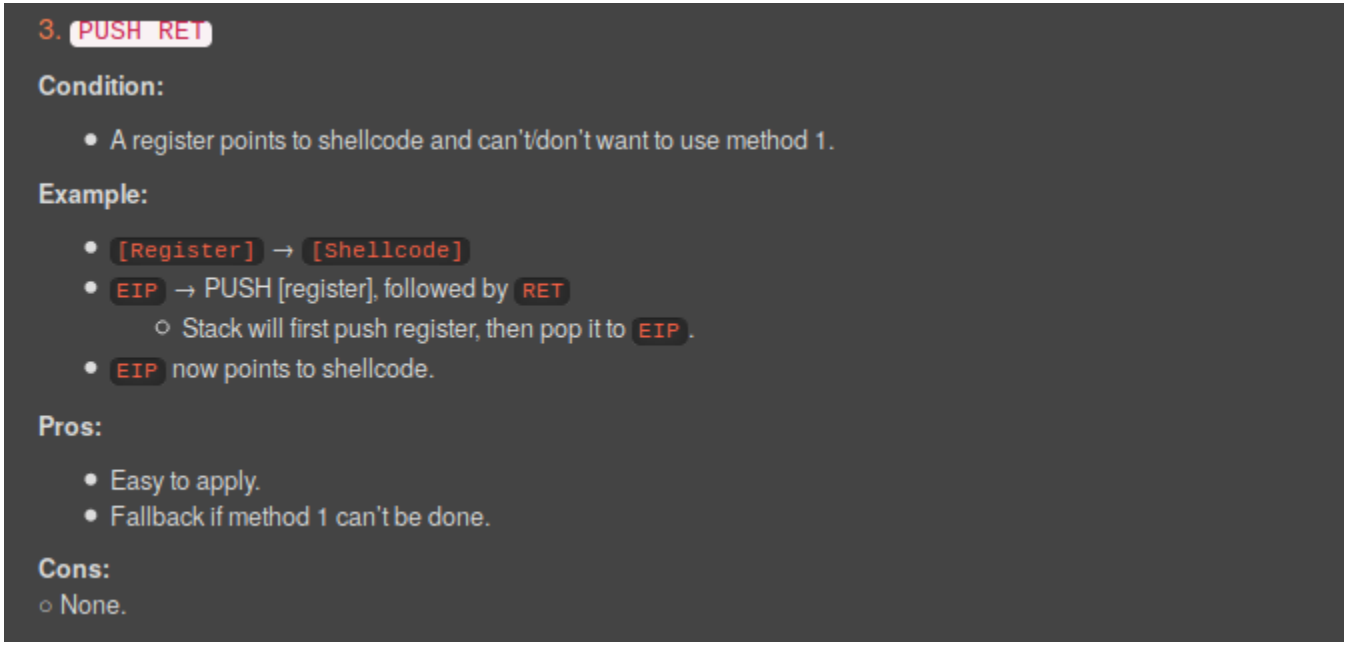
Cons:

- Less frequent pattern.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- push return.



3. `PUSH RET`

Condition:

- A register points to shellcode and can't/don't want to use method 1.

Example:

- `[Register]` → `[Shellcode]`
- `EIP` → PUSH [register], followed by `RET`
  - Stack will first push register, then pop it to `EIP`.
- `EIP` now points to shellcode.

Pros:

- Easy to apply.
- Fallback if method 1 can't be done.

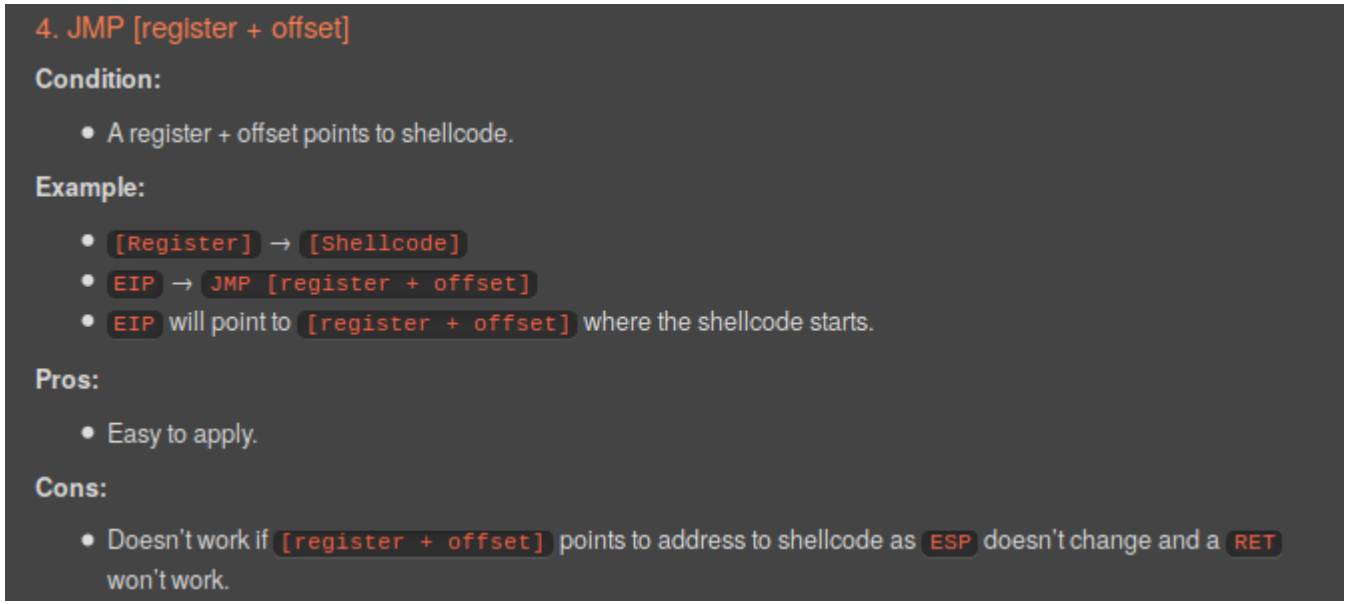
Cons:

- None.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- jmp [reg + offset].



4. JMP [register + offset]

Condition:

- A register + offset points to shellcode.

Example:

- `[Register]` → `[Shellcode]`
- `EIP` → `JMP [register + offset]`
- `EIP` will point to `[register + offset]` where the shellcode starts.

Pros:

- Easy to apply.

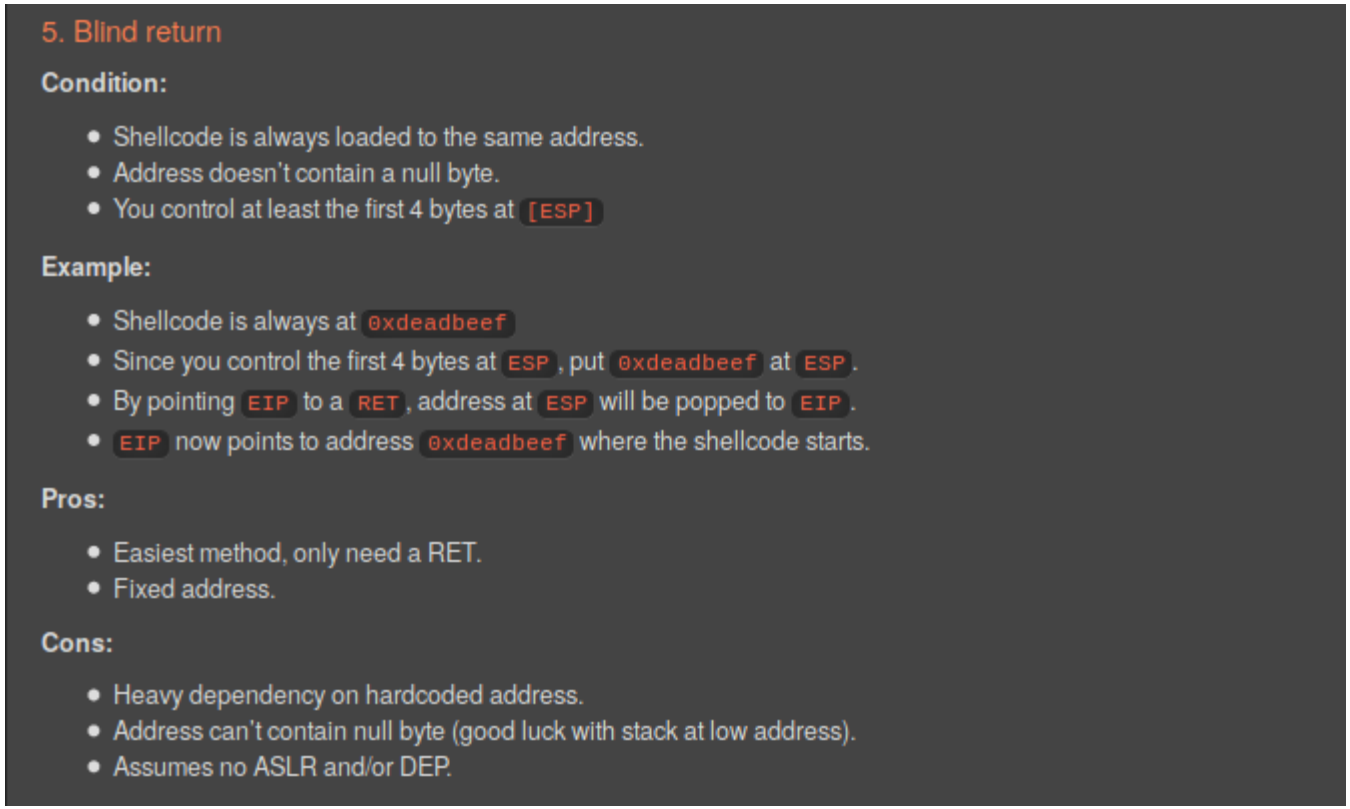
Cons:

- Doesn't work if `[register + offset]` points to address to shellcode as `ESP` doesn't change and a `RET` won't work.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- blind return.



5. Blind return

Condition:

- Shellcode is always loaded to the same address.
- Address doesn't contain a null byte.
- You control at least the first 4 bytes at `[ESP]`

Example:

- Shellcode is always at `0xdeadbeef`
- Since you control the first 4 bytes at `ESP`, put `0xdeadbeef` at `ESP`.
- By pointing `EIP` to a `RET`, address at `ESP` will be popped to `EIP`.
- `EIP` now points to address `0xdeadbeef` where the shellcode starts.

Pros:

- Easiest method, only need a RET.
- Fixed address.

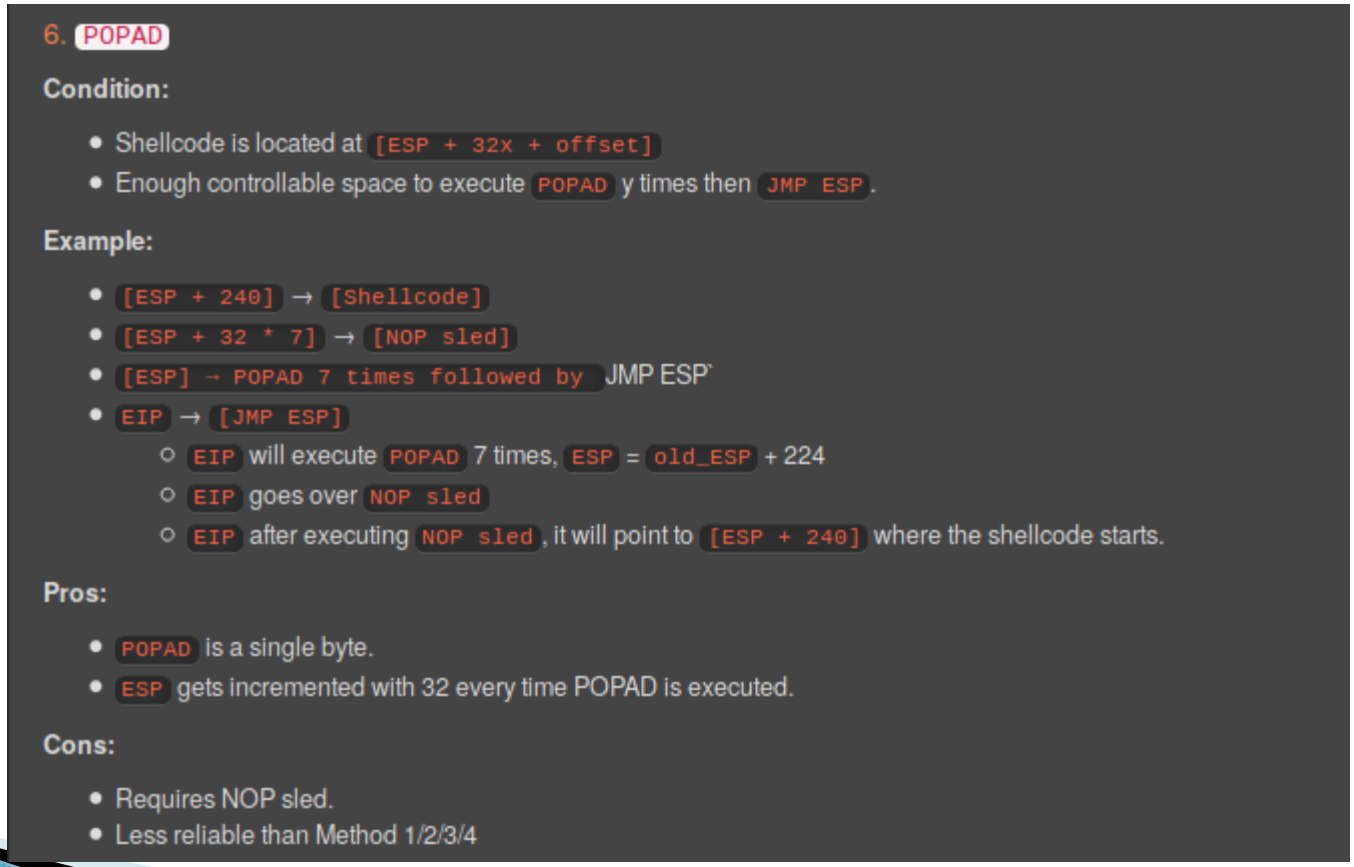
Cons:

- Heavy dependency on hardcoded address.
- Address can't contain null byte (good luck with stack at low address).
- Assumes no ASLR and/or DEP.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- popad.



6. `POPAD`

Condition:

- Shellcode is located at `[ESP + 32x + offset]`
- Enough controllable space to execute `POPAD` y times then `JMP ESP`.

Example:

- `[ESP + 240]` → `[Shellcode]`
- `[ESP + 32 * 7]` → `[NOP sled]`
- `[ESP]` → `POPAD 7 times followed by` JMP ESP`
- `EIP` → `[JMP ESP]`
  - `EIP` will execute `POPAD` 7 times, `ESP` = `old_ESP` + 224
  - `EIP` goes over `NOP sled`
  - `EIP` after executing `NOP sled`, it will point to `[ESP + 240]` where the shellcode starts.

Pros:

- `POPAD` is a single byte.
- `ESP` gets incremented with 32 every time POPAD is executed.

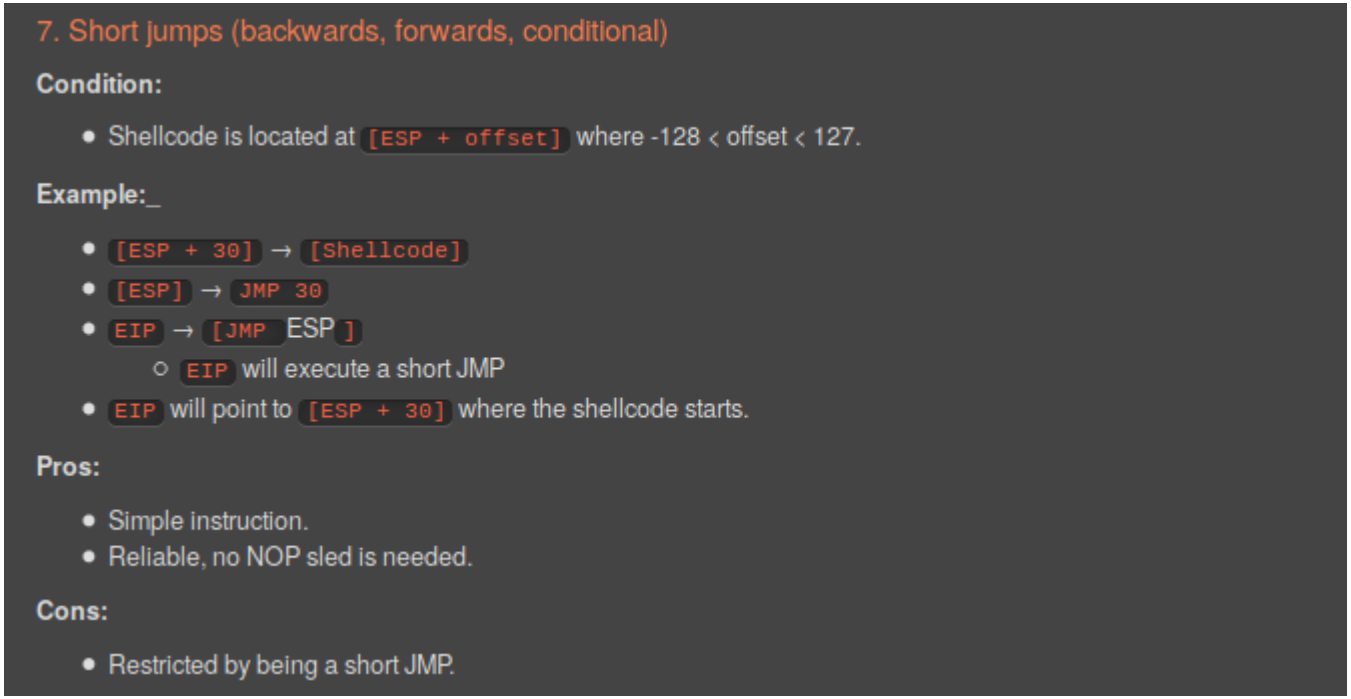
Cons:

- Requires NOP sled.
- Less reliable than Method 1/2/3/4

https://www.abatchy.com/2017/05/jumping-to-shellcode.html

Jumping to Shellcode

- Backward jumps.



7. Short jumps (backwards, forwards, conditional)

Condition:

- Shellcode is located at `[ESP + offset]` where -128 < offset < 127.

Example:_

- `[ESP + 30]` → `[Shellcode]`
- `[ESP]` → `JMP 30`
- `EIP` → `[JMP ESP]`
  - `EIP` will execute a short JMP
- `EIP` will point to `[ESP + 30]` where the shellcode starts.

Pros:

- Simple instruction.
- Reliable, no NOP sled is needed.

Cons:

- Restricted by being a short JMP.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode

- Jump ESP.

8. Hardcoded address

Condition:

- Shellcode is always located at specific address.

Example:

- `0xdeadbeef` → `[shellcode]`
- `EIP` → `JMP ESP`
- `ESP` → `JMP 0xdeadbeef`

Pros:

- Simple instruction.

Cons:

- Unreliable.
- Address can't contain null bytes.

https://github.com/[redacted]batchy17/ExploitDevSnippets/tree/master/Corelan

Jumping to Shellcode : Windows SEH

- Exception Registration Record.

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

- Pointer to exception handler function.

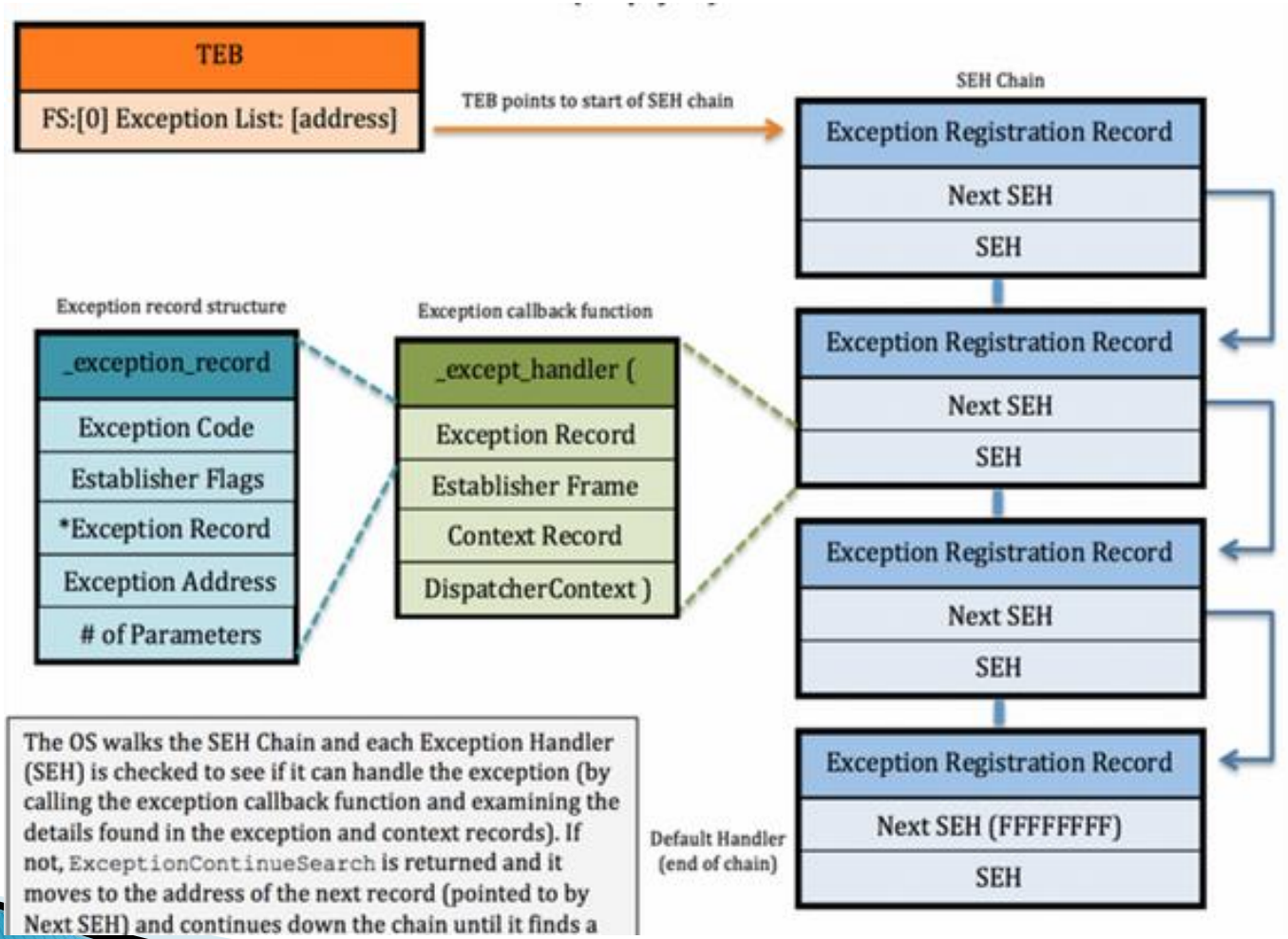
```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    oid EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
);
```

Jumping to Shellcode : Windows SEH

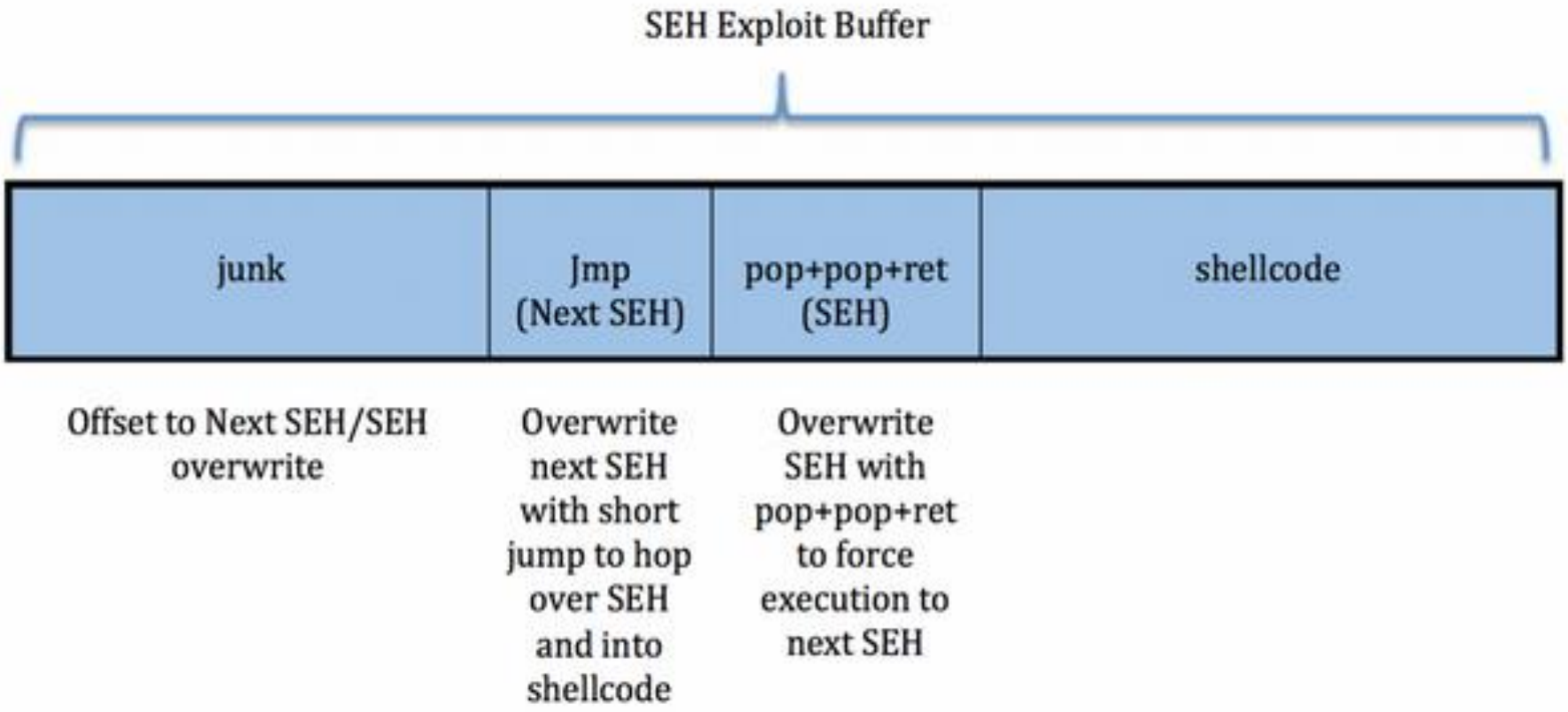
- Exception Record.

```
typedef struct _EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct _EXCEPTION_RECORD *ExceptionRecord;
        PVOID ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Jumping to Shellcode : Windows SEH



Jumping to Shellcode : Windows SEH



Summary

- Basic binary exploitation model.
  - Buffer overflow.
- Bypassing ASLR.
- Other stack attacks.
  - Format string vulnerabilities.
  - Integer overflows.
- Heap overflows.
- Hardware side channels.
  - Effective due to lower frequency of hardware updates.

Resources

- “Return-Oriented Programming: Systems, Languages, and Applications” by RYAN ROEMER, ERIK BUCHANAN, HOVAV SHACHAM and STEFAN SAVAGE University of California, San Diego.
- https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
- http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf
- Bypassing browser memory protections in Windows Vista by A Sotirov, M Dowd - Blackhat USA, 2008.
- https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

That’s for the classes