

Memory Exploitation

UDURRANI

I have divided this project in 2 phases.

- Exploit

- Malware (Not in this writeup)

I will cover exploit first since not many security products deal with exploits. They are mostly focusing on malware. Keep in mind you don't need to master all the concepts. Just a few fundamental concepts so you get a feel of how things work. Before we really get into exploit, lets cover few things that would help us as we move forward.

To understand Security, you should understand Computer architecture, especially Memory and CPU.

****CPU**:**

We call it the centerpiece / the brain of the computer. It consists of transistors and channels through which current flows. Every type of CPU has its own language i.e. machine language. Machine language is very complex in nature. That's why we have programming languages, where a compiler or an assembler converts a program to machine language. This component takes care of executing instructions. Let's say instructions of a currently loaded program / process.

****Instruction???**

Instruction is a sequence of 0's and 1's that describes an operation e.g. ADD or COMPARE. Instructions could be carried out in multiple ways: Read, Write, Calculate, Jump , Add numbers, Test, Signals etc.

Instructions could have data they act on, which is stored in a register. This is the fastest way for CPU to access data even though CPU has limited number of registers. A register is a storage device that can "store," or "remember a single word. Its very important to understand this distinction i.e. instructions (code) and data. Data normally flows as variables in a programming language. If I ask you to add 10 to another integer value e.g. 10 + 30. What does that say? 10 and 30 represents data and ADD would represent an instruction. Computers take this sort of distinction very seriously. We will get into it later but for now just remember data and code.

This data has to flow from storage to cpu through some sort of a transmitting means. This is done via BUS. An instruction is represented as a binary, 16, 32, or 64 bits wide. Before execution it must be decoded. This is handled by Control Unit.

****So Lets remember:****

- 🔊 CPU's life is to fetch, decode and execute. In some cases it can store.
- 🔊 Data & code separation
- 🔊 Transmission is done via a BUS
- 🔊 Data & Code creates the actual output i.e. the execution flow.

I just mentioned fetch and execute. Where does it fetch from? Answer is simple, the storage. Didn't I say that registers are used for storage? indeed! But registers are limited and I would like to put more data some where. What about the main memory i.e. RAM? Thats right. RAM can store much more. This means CPU can fetch instructions from the memory and execute them. CPU is the biggest consumer of computers main memory.

****MEMORY**:**

This component is responsible for storing data and instructions. Unlike registers memory is not as close to the CPU. Registers are great if one has to use a data value quiet often in the program. Its not necessary that instructions or data should be adjacent to each other, it could be completely random, hence random access memory (RAM). Memory is used to hold data and software for the processor.

Memory follows a concept of flip-flop. Memory holds a value of 1 when capacitor is charged and 0 when dis-charged. Capacitor is like a bag that holds or stores charge. In other words it holds electrons. To store 1, electrons move towards the bag i.e. bag is filled with electrons. To store 0, its emptied out. You may ask who will charge and dis-charge? CPU and memory controller (Please lookup memory controller). Transistor and a capacitor are paired to create a memory cell, which represents a single bit of data. The capacitor holds the bit of information -> a 0 or a 1. Transistor doesn't store data, its used for signal amplification.

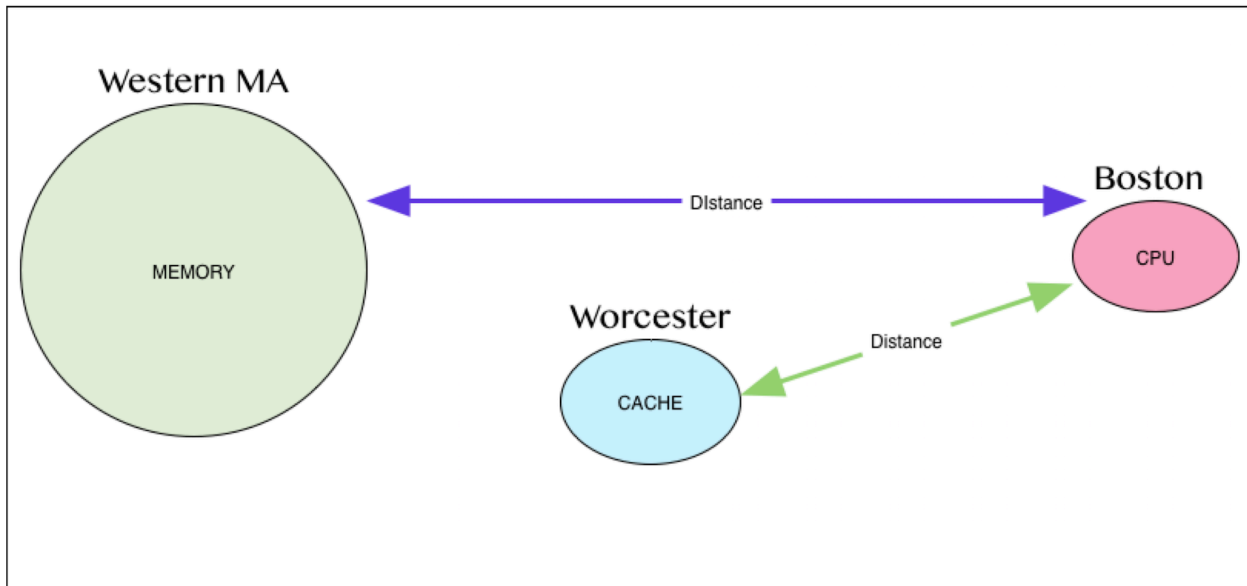
Enough about electronics :)

The memory holds two types of information: data items and programming instructions. The two types of information are usually treated differently, and in some computers they are stored in separate memory units. Memory location is identified by an address i.e. if you would like to get to a location you need that address. To speed things up another component called slave memory or cache memory could also be used. Let me try and put more sense into it.

Lets assume, CPU is a Small property in Boston MA, However CPU has very limited space, as property in Boston is very expensive. So we buy a larger property outside of Boston, some where in Western MA, where we can store a lot of items. (Let's call this new property in Western MA, memory). Property in Western MA is much cheaper than Boston but for CPU to get items from western MA (memory) could take a long time.

What do we do?

We need to buy a small place where we can put some of the items that CPU require more frequently. We can buy another property somewhere between Boston (CPU) and Western MA (MEMORY). Let's say the 3rd property is in Worcester MA. Worcester MA is the cache :)



CPU (Boston) needs an item called foo. CPU (Boston) calls the cache (Worcester) and asks for foo?

CPU (Boston): Is foo available?
 Cache (Worcester): Yes (This equals cache hit)

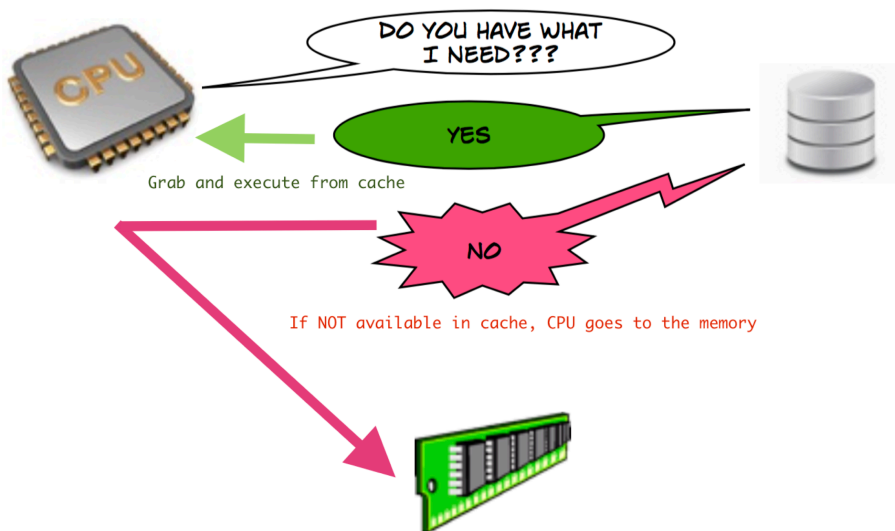
CPU(Boston): Is foo available?
 Cache(Worcester): Nope! (Cache Miss)
 CPU is not very happy about it because it has to go all the way to Western MA to get foo. Traffic conditions ... Undetermined!
 CPU(Boston): gets foo. At the same time it keeps one copy of foo in the cache for next time.

Worst case, even memory doesn't have it, this is really bad because now it has to be paged-in from the SLOW disk.

I hope that made sense?????

FAST -- TO -- SLOW


CPU <--> L1 Cache <--> L2 Cache <--> Memory <--> Disk




****STACK****

The program stack is an area of memory that supports the execution of functions. When a function is called, a stack frame is created and pushed onto the program stack. When function terminates, stack frame is popped off the stack. A stack pointer usually points to the top of the stack. A stack base pointer (frame pointer) is often present and points to an address within the stack frame, such as the return. The stack buffer has an associated data item, the stack pointer, SP, which contains the address of the top of the stack. This address is of course increased (or decreased) when data is pushed (or popped). The addresses of the different data items stored in the stack are simply determined by their distances (offsets) to the stack pointer. When `main()` starts, its local variables are allocated in the stack. When the function returns, it destroys its local variables and the function arguments by popping them from the stack.

A data element is placed on top of the stack by a **PUSH** instruction; a data element is removed from the top of the stack by a **POP** instruction if the operand size is 16 bits, then the SP register is incremented/decremented by 2. If the operand size is 32 bits, then the ESP register is incremented/decremented by 4

 push instruction will decrement SP

 pop instruction will increment SP

```
push ebp // Stack pointer points to the top of the stack
```

```
mov ebp, esp // make ebp new esp
```

****Heap****

When memory is dynamically allocated, it comes from the heap. Heap memory is dynamically allocated at run-time by the application. Let's allocate some memory to foo.

```
char *foo = (char *)malloc(30) // foo in this case is a pointer.
```

Allocated memory on the heap **MUST** be released or freed after use. To free above allocated memory we use free function call.

```
free(foo)
```

Linked lists and queues are enabled using heap section of the memory.

Program Counter / Instruction Pointer:

PC or EIP | IP | RIP plays an important role. CPU includes a program counter whose output is interpreted as the address of the instruction that should be executed next in the current program. This way CPU knows what instruction to execute next. This is a very important concept so please make sure you do more research on this one.

A little about registers: A register is a storage device that can store or remember a value. E.g

- General and data-segment addressing: , EAX, EBX, ECX, ESP, EBP, ESI, EDI**
- Segment registers: CS, DS, ES, SS**

I would suggest to look them up and get some info regarding registers.

VULNERABILITY:

Programming is an art. Some developers are good at it and some are not. Either way they make mistakes, even the best of developers. Writing code is no joking matter. Most production code has to interface with other code, other libraries, files, memory etc. This means data is passed between all these components. If this data is passed incorrectly or handled incorrectly, this could have shitty results. If I ask you to have a buffer that can hold 10 bytes

Using a language like C or C++ you would do something like.

```
char array[10] // Where variable name is array.
```

What if you try to put data into this array that exceeds the actual size of buffer??? What happens to the exceeded data??? In this situation extra data can overflow into adjacent memory locations corrupting valid data. This sort of a behavior is called Stack Overflow. Important thing to remember is: Stack cannot handle this situation and could lead to change of execution flow of the program. Program itself would have no clue what the hell is going on. Changing program control flow execution is a dangerous matter.

Overflow can happen in heap region as well, allowing an attacker to overwrite heap-stored data. If allocated region of the memory is freed or deallocated more than once, could cause code injection. Another example could be use after free.

```
char *foo = (char *)malloc(10)  
free(foo);
```


What if same memory region is used again i.e after free'ing it or for some reason it went out of scope. You may ask how would it get out of scope if we haven't free'd the memory? On the other hand, what if you don't free the memory? What would happen? In that situation you can run into a memory leak.

E.g. if we allocate memory and never free it, it could cause the machine to reboot or result in an exploit.

```
for(b=0;b<=10;b++){
printf("Allocating ...\n");
char *a = (char *)malloc(100);
}
```

In this case I have 11 blocks and allocated 1110 bytes without freeing them.

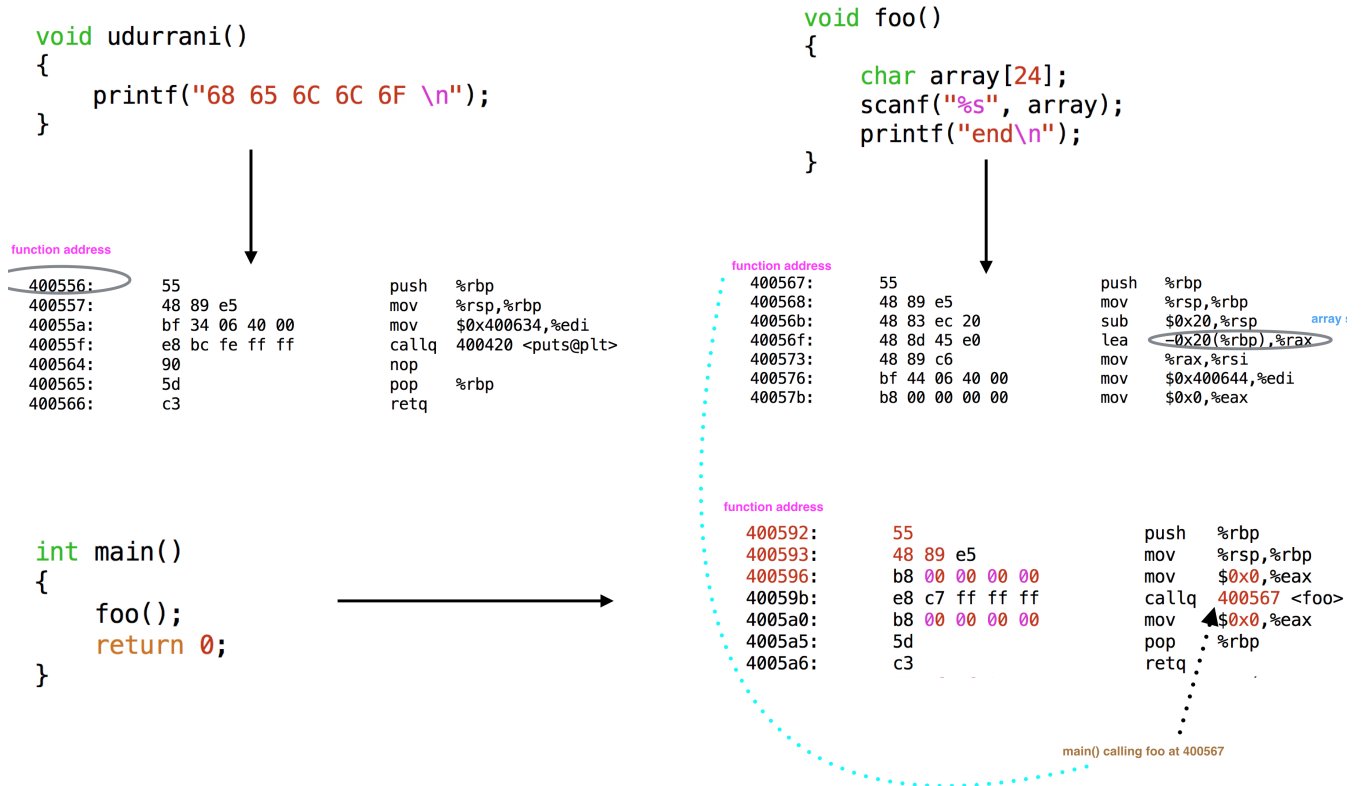
In C++ we use:

```
myClass * foo = new myClass;
```

Later the memory could be free'd by using.

```
delete(foo);
```

Let's look at a simple buffer overflow. We have total of 3 functions: udurrani(), foo() and main()



Code Flow: main() calls foo() only, it doesn't call udurrani().
ISSUE: foo() doesn't do any size validation i.e. if array is 24 bytes or not
Question: What if user provides more than 24 bytes???

Can this behavior write to adjacent memory location? Can we change the flow and call the next function i.e. udurrani()???

If I run this code and provide input that is size of(array), this will call the function foo().
 foo() runs and gives control back to main()
 main() ends and gives control back to the OS.

I am using 64 bit arch where each register is 8 bytes. In function foo() we can say:

Registers + Buffer + Return = 40 bytes.

Registers = RBP, RIP, RSP etc

Its time to overRun the buffer and overwrite the return address.

Lets write 40 characters e.g. the char 'A' followed by the address of udurrani().

(AA) - (400556)

Since its a hex address, it should be \x40\x05\x56

Please make sure you know about the endian-ness i.e. you are using little endian or big endian arch. Endian-ness is the way your computer store words in the memory. I am using little endian so let's reverse this address

Little-endian = Least significant byte gets stored in the smallest address.

(AA) - (\x56\x05\x40)

You can run this using any scripting language or run it manually.

Result: Function udurrani() gets called.

```

end
68 65 6C 6C 6F

```

← foo() prints the string end
 ← udurrani() prints hex value that says hello

If I further debug it or get a backTrace, will show something like the following. NOTE: 41 in hex = letter 'A'

```

Jump to the invalid address stated on the next line
at 0x4141414141414141: ???
by 0x4141414141414140: ???
0x4141414141414141 in ?? ()
0x0061414141414141 in ?? ()
0x0000000100000000 in ?? ()
0x0000000004005d0 in ?? ()

```

Summary: Bounds checking is very critical when you write programs. Going out-of-bounds leads to un-known behavior and attackers can take advantage of such scenarios. In such conditions program itself has no clue whats going on. Buffer Overflow can lead to

- Denial of Service (Critical in case of servers e.g. Web, FTP, DB servers)
- Subversion of execution flow (Very critical since this could result in arbitrary code execution)

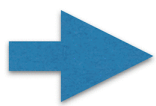
Dangling pointers: pointers that points to invalid data

Consider the following as a function thats trying to return a string
 Let's name it func()

```

char array[10];
strcpy(array,"udurrani");
return(array);

```



```

function address
4004e6: 55          push  %rbp
4004e7: 48 89 e5    mov   %rsp,%rbp
4004ea: 48 8d 45 f0 lea  -0x10(%rbp),%rax
4004ee: 48 ba 75 64 75 72 72 movabs $0x696e617272756475,%rdx
4004f5: 61 6e 69
4004f8: 48 89 10    mov   %rdx,(%rax)
4004fb: c6 40 08 00 movb  $0x0,0x8(%rax)
4004ff: b8 00 00 00 00 mov  $0x0,%eax
400504: 5d          pop   %rbp
400505: c3          retq

```

Let's call the function func()
 E.g. char *a = func()

In this situation, nothing is returned???

Because we are allocating string on the stack, and then returning a pointer to it. When the function returns, any stack allocations become invalid, the pointer now points to a region on the stack that is likely to be overwritten the next time a function is called. Once the function returns values are popped and the calling function won't get anything or pointer would point to some junk in the memory. Hence pointer pointing to invalid data.

```

55          push  %rbp
48 89 e5    mov   %rsp,%rbp
48 83 ec 10 sub  $0x10,%rsp
b8 00 00 00 00 mov  $0x0,%eax
e8 ce ff ff ff callq 4004e6

```

main() calling func() address 4004e6

In this situation: Either allocate the string on the stack on the caller side and pass it to your function:

We can also look at allocation on heap.

If we allocate 10 bytes on heap.

```
char *var = (char *)malloc(10);  
If you assign var to anything e.g. var = "udurrani";
```

What happens if var is free'd, destroyed, deleted or it goes out of scope?

```
free(var)
```

In this situation var is on stack and the memory allocated is on the heap. At the same time its pointing to invalid memory i.e. its already free'd or went out of scope. Let me try to re-use it again: printf("%s\n", var);

Here var becomes a dangling pointer, its pointing to invalid memory.

```
Address for free() 400430:    ff 25 52 05 20 00    jmpq  *0x200552  
400436:    68 00 00 00 00    pushq $0x0  
40043b:    e9 e0 ff ff ff    jmpq  400420  
  
400576:    55                push  %rbp  
400577:    48 89 e5          mov   %rsp,%rbp  
40057a:    48 83 ec 10      sub  $0x10,%rsp  
40057e:    bf 0a 00 00 00  mov  $0xa,%edi  
400583:    e8 d8 fe ff ff  callq 400460 .....▶ malloc returns a pointer to memory in rax  
400588:    48 89 45 f8      mov  %rax,-0x8(%rbp)  
40058c:    48 c7 45 f8 44 06 40  movq $0x400644,-0x8(%rbp)  
400593:    00  
400594:    48 8b 45 f8      mov  -0x8(%rbp),%rax  
400598:    48 89 c7          mov  %rax,%rdi  
40059b:    e8 90 fe ff ff  callq 400430 .....▶ Free'ing the memory  
4005a0:    48 8b 45 f8      mov  -0x8(%rbp),%rax  
4005a4:    48 89 c7          mov  %rax,%rdi  
4005a7:    e8 94 fe ff ff  callq 400440 .....▶ Using the free'd memory again  
4005ac:    b8 00 00 00 00  mov  $0x0,%eax  
4005b1:    c9                leaveq  
4005b2:    c3                ret
```

This situation could also be called use after free (UAF)

Vulnerability could happen in a lot of ways e.g. string formatting, lack of input validation and bounds checking, integer errors, dereferencing a pointer that points to an invalid memory location etc. I will cover more later.

Function Pointers:

Pointer itself is a very important feature of programming. Imagine having a huge data structure that you have to pass to other functions. It would make it much more efficient if you pass the address to that data structure isn't it? Well if you'd like you can keep passing the whole data structure but I won't recommend. It makes it easier to pass large block of memory to functions. It also makes thing more speedy.

You can also get an address of a function and use it as call back. You can pass a function to another function.

Functions are pointers that points to a block of code. You can also have a function pointer i.e. a pointer pointing to a function. Why oh why do I do that?

Call backs to another functions

You use any object oriented language like Ruby or Python where you never have to worry about stack or the heap or dynamic allocation? In the background you use function pointer to simulate objects and classes. A major use case is in asynchronous programs as well, where one can call a function and pass it a function pointer

Look at the the following function.

```
void functionB(int a)  
{  
    printf("%d\n", a);  
}
```

Lets make a function pointer

```
void (*foo)(int);  
foo=&functionB;
```

How do I call it?

```
(*foo)(12);
```

```
push  %rbp  
mov   %rsp,%rbp  
sub  $0x10,%rsp  
movq  $0x40055c,-0x8(%rbp)  
mov  -0x8(%rbp),%rax  
mov  $0xc,%edi  
callq **rax ← Function Pointer  
mov  $0x0,%eax  
callq 40054c ← Normal function called by its address
```

The only reason I covered function pointers here is because exploits normally either overwrite return address (stack) or function pointers (heap). We will cover that in-depth later. This paper is only to cover basic stuff. In the second phase I will cover more about function pointers.

Exploitation:

I am sure you must be thinking why talk about vulnerability for so long. The answer is very simple, no vulnerability means no exploit. You need a bug in your software/application. Even though all bugs cannot turn into exploits but all exploits are result of a software bug. When attacker is writing an exploit the first rule is to trigger the vulnerability.

FIRST STEP: Trigger the BUG

After triggering the bug, it only gets more complicated :) You need to find the code you want to execute and its address! If you remember we talked about data and instruction (code) separation. Code section in memory is Read and Execute. Data means Read and Write. In memory we have stack, heap, data and code sections. Heap and Stack sections are not executable. When you use things like `int a = 0;` this goes on the stack. For dynamic memory allocation like `malloc()`, heap kicks in.

CPU fetches an instruction and executes it but it can ONLY be executed if it has an execute bit on. Its called NX bit. This is how OS can prevent basic exploitation attacks. We call this feature DEP (Data execution prevention). This literally means execute code not data.

DEP is a mitigation offered by the OS.

If somehow you got passed DEP, you need to know the address of the code. All OS's will load functions, libs etc at a random address. If address is random,

attacker can't just land an attack. Attacker has to somehow know the address or by-pass that layer of randomization. This one is called ASLR (address space layout randomization).

ASLR is another layer of mitigation provided by the OS. What about prevention against stack overflows? OS has that one as well, its called GC cookie or a canary. Some of the OS mitigations: DEP, ASLR, GS Cookie etc etc.

Question: *If OS has all the mitigation, how would an attacker launch an exploit???*

Attackers will use a specific technique to by-pass OS's mitigation.

What are some of the techniques???

ROP, JIT & Heap Spray, Stack pivot etc ...

If you have an application, third party or developed in-house. That application is made according to a certain specification. E.g.

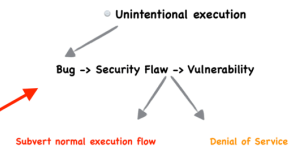
- Log-in
- Credentials
- Welcome Page
- Error Code
- DataBase Interface in the background
- Modify or create new documents
- Save or submit.

This flow is called intentional execution.



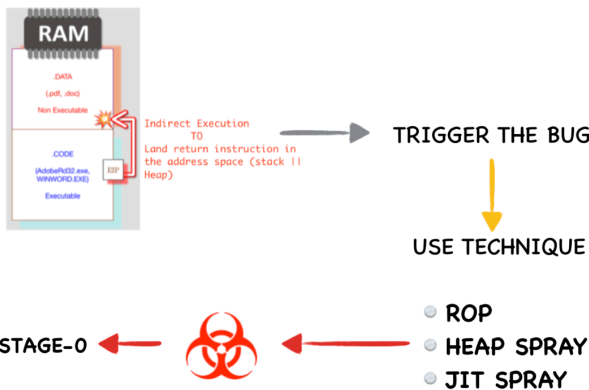
• Intentional Execution

• Unintentional execution???



Is it possible for me to divert this execution flow? Hence un-intentional execution? To achieve such behavior attacker has to somehow smuggle exploit + malicious code within the data section of the memory E.g: attacker may put together a specially crafted PDF document and send it to the victim. In this case vulnerable application is acrobreader (version N). Now all attacker has to do is use a technique to achieve indirect execution i.e. execute the code that exists in the data section.

Simple flow of stack overflow.



Normally this is not as straight forward.

For an overRun exploit:

- > Buffer Overflow (Remember you have to trigger the bug)
- > This overRun will Overwrite the base pointer, Instruction pointer is next
- > JMP is used here
- > Payload

Somewhere in there you need to by-pass OS exploit mitigation

AT this stage you are compromised. BARA BING BARA BOOM

Recap

- ☑ Find a vulnerability
- ☑ Trigger the bug / security flaw
- ☑ Use a technique to by-pass OS mitigation
- ☑ Make the malicious code (on stack or heap) executable
- ☑ Launch the malicious code
- ☑ Compromised!!!

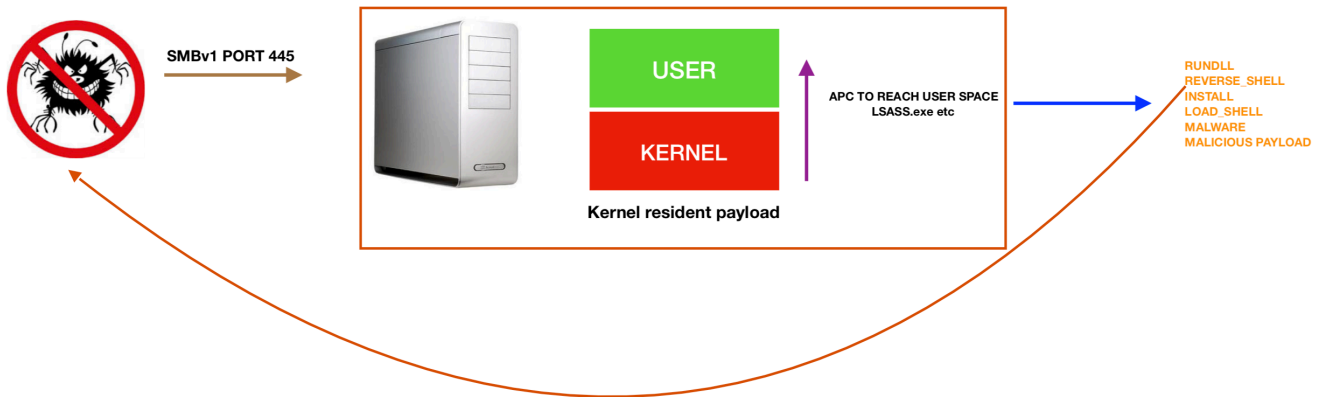
Here is a very simple flow (**Keep on clicking**)

<http://udurrani.com/exp0/Vulnerabil/index.html>

Exploits have different kinds but memory corruption exploits are very complex. It only gets more complex when we get into OS / Kernel level exploits. If you remember WanaCry (If you don't then go to <http://udurrani.com/0fff/all1.pdf>), propagation was done by the exploit. The most critical component of the whole campaign was the exploit. Without the exploit, there wouldn't have been any propagation or lateral movement. No privilege escalation either. The exploit mainly used SMBv1 transactions to perform read / write IO's between client and server. If the request size > SMB_MAX_BUFFER_SIZE, remaining bytes are processed by trans2 request. The attacker has embedded the exploit code within that large data. Attacker has to trigger the vulnerability to hijack the flow.

NOTE: The word PROCESS / PROCESSING is the key. Attacker smuggled in the bad shit code in the data and someone's gotta process that data, hence the exploit, **GOT IT????**

Once the payload is at the kernel layer (SMB is processed by the kernel), the payload uses APC to launch the backdoor in the userSpace and then wanaCry starts its magic.



Let's focus more on this lateral movement: If we remove the exploit part from wanaCry, there is no lateral movement or privilege escalation. An infected machine can copy the payload to other machines i.e. internal or external. Let me show you how the payload tried to scan external ip addresses.

<http://udurrani.com/0fff/capt/a.html>

<http://udurrani.com/0fff/capt/b.html>

<http://udurrani.com/0fff/capt/c.html>

Zero-day OS / kernel exploits are not easy to detect by any security provider.

