# Buffer Overflow attacks and Related worms

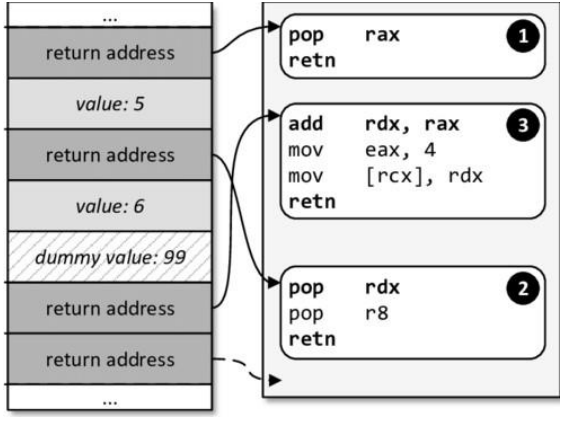
Karan Agarwalla : 180050045
Mohammad Ali Rehan : 180050061
Neel Aryan Gupta : 180050067
Shreya Pathak : 180050100

# Introduction

The execution stack is an indispensable component of modern operating systems and architectures. We have studied out lack of securities and use of unchecked buffers can be successfully used to take control of any machine or server and gain root access. We also present case studies of how this technique has been used in the past for mass scale attacks leading to DDoS and infiltration of user systems across the globe.

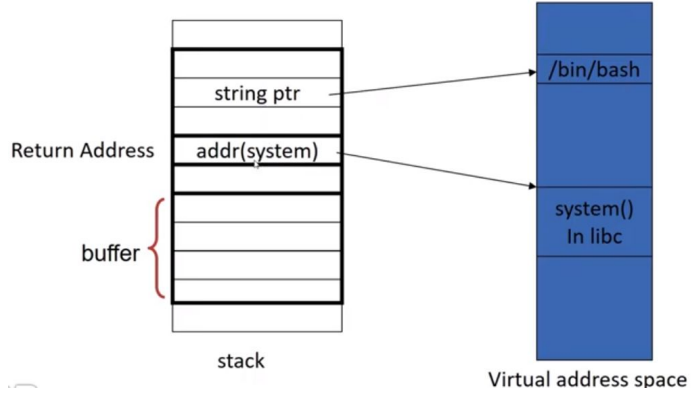
# ret2sysall

We make use of the fact that libraries are statically linked and so their virtual address are always same. We use a tool like *gef* to find gadgets in the code and library. Gadgets help us to execute simple instructions like *move*, *pop* and then return to stack. We design a chain of address of gadgets, which can be executed one after the other and call shell in the end.




# ret2libc

In ret2libc, the dynamically linked library has fewer gadgets preventing us from executing a ret2syscall attack. We first try to decipher the libc base address by leaking the GOT table entry value for a known function using *puts()*. The return address placed is same as *vuln()* allowing us to overflow the buffer in the *puts()* function call again. With this libc address, call the *system('/bin/sh')* using offsets of functions from libc.



# ret2dlresolve

In ret2dlresolve attacks, the library is dynamically linked and there are no *puts()* function calls preventing us from executing either of ret2sycall and ret2libc attacks. We proceed to replace the *read()* entry value in the GOT with that of *system()* function.

To do this we first forge the different structs involved in fetching symbol information by *runtime_resolver* and place them in *bss* segment. We also place strings *system* and */bin/sh* in the *bss* segment. Then we call the *runtime_resolver* with appropriately forged arguments to replace the GOT entry of a function(here *read()*). The call to resolver automatically executes our replace function(here *system()* whose argument */bin/sh* was put on the stack) as it thinks this function/symbol has been called for the first time in the code.

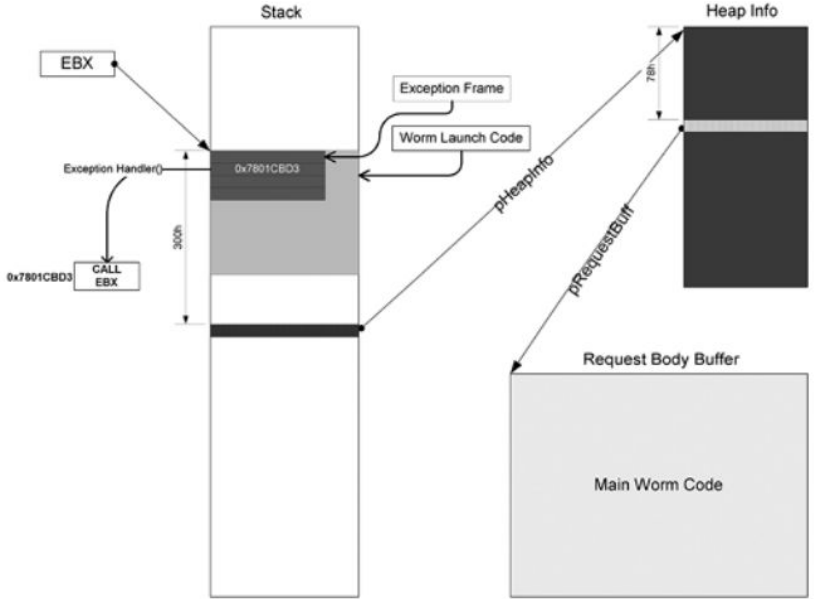
# Morris Worm

Morris Worm was launched in the 80s and employed several techniques to leak sensitive data. One technique targeted the finger daemon which was run many servers for client information and had an unchecked buffer, which was they key to the attack. The  worm overflowed the 512B buffer and overwrote the return address on the stack to execute malicious code placed on the stack by the worm. Below is the content it placed in the buffer to make the system call:

```
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
```

# CodeRed Worm

The Worm exploited unchecked buffers present in the dynamic library *idq.dll*. This was used to receive HTTP GET requests from users and allows users to search for documents by keyword. The worm sent a cleverly crafted string as URL to overwrite the address in the trapframe and make the EIP point to the stack. This URL also doubled up as instructions for launching other features like spreading further, or installing trojans.




```
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
```

# Challenges Faced & Takeaways

- The string for correct overflow and execution redirection must be very carefully crafted after taking into account the underlying ISA, architecture(32bit or 64bit) and endianness.
- The assembly source code for CodeRed wasnt freely available so we had to recreate it from the URL using tools to get x86 instruction from the machine code.
- The BSD Unix 6 which the morris worm attacked, has been patched up. So it was not straightforward to figure what was the exact functionality that was being attacked, however some archives and reading the source code of relevant portion helped us find it.

- ❖ Through this project, we gained a deeper understanding of how the execution stack can be particularly dangerous when enough protections are not used
- ❖ As programmers, it teaches how one must be always careful in designing interactive programs and libraries. Even a single flaw is enough to infiltrate a machine
- ❖ We learnt about the many security provisions that are in place like NX bit, canary etc. to prevent successful execution of malicious code
- ❖ As a side, we also got to explore other related topics like DDoS or dynamic linking.