Binary Exploitation 2a

Return Oriented Programming & ASLR

Milan Patnaik

Indian Institute of Technology Madras

Where Are We ?

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.
- Non Executable, No Canaries and No ASLR.
  - Overwrite the return address.
  - Return to libc restricted by system().

Where Are We ?

- Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Shellcode in stack.
- Non Executable, No Canaries and No ASLR.
  - Overwrite the return address.
  - Return to libc restricted by system().
- Non Executable, No Canaries and No ASLR.
  - Overwrite return address.
  - Return Oriented Programming.
  - Execute arbitrary code.

Return Oriented Programming (ROP)

Return Oriented Programming

Not secure | cve.mitre.org/cgi-bin/cvekey.cgi?keyword=return-oriented+programming+%28ROP%29

Common Vulnerabilities and Exposures

CVE List CNAs WGs Board About News & Blog

NVD
Go to for:
CVSS Scores
CPE Info
Advanced Search

Search CVE List Download CVE Data Feeds Request CVE IDs Update a CVE Entry

TOTAL CVE Entries: 122277

HOME > CVE > SEARCH RESULTS

Search Results

There are **2** CVE entries that match your search.

Name	Description
CVE-2018-5392	mingw-w64 version 5.0.4 by default produces executables that opt in to ASLR, but are not compatible with ASLR. ASLR is an exploit mitigation technique used by modern Windows platforms. For ASLR to function, Windows executables must contain a relocations table. Despite containing the "Dynamic base" PE header, which indicates ASLR compatibility, Windows executables produced by mingw-w64 have the relocations table stripped from them by default. This means that executables produced by mingw-w64 are vulnerable to return-oriented programming (ROP) attacks. Windows executables generated by mingw-w64 claim to be ASLR compatible, but are not. Vulnerabilities in such executables are more easily exploitable as a result.
CVE-2013-6791	Microsoft Enhanced Mitigation Experience Toolkit (EMET) before 4.0 uses predictable addresses for hooked functions, which makes it easier for context-dependent attackers to defeat the ASLR protection mechanism via a return-oriented programming (ROP) attack.

BACK TO TOP

Return Oriented Programming Attacks

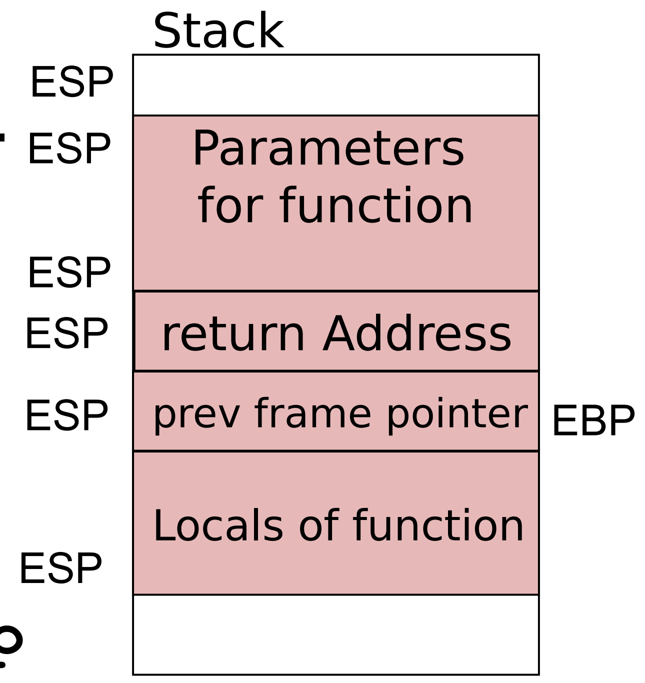
- Discovered by Hovav Shacham of Stanford University
- Subverts execution.
  - As with the regular ret-2-libc, can be used with non executable stacks since the instructions can be legally executed.
  - Unlike ret-2-libc does not require to execute functions in libc (can execute any arbitrary code).

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

Stack : Function Call

Call instruction has 2 steps:

- Push the contents pointed to by EIP.
- Decrease ESP by 4 (32bit machine)



In main

```
push $3
push $2
push $1
```

In function

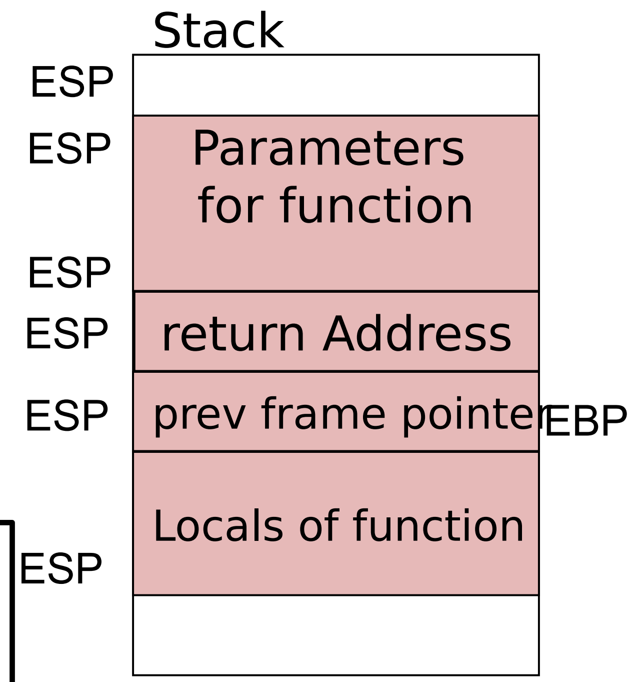
```
push %ebp
movl %esp, %ebp
sub $20, %esp
```

%ebp : Frame Pointer
%esp : Stack Pointer

Stack : Function Return

Ret instruction has 2 steps:

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



In main

```
push $3
push $2
push $1
```

In function

```
movl %ebp, %esp
```

Action by leave instruction

%ebp : Frame Pointer
%esp : Stack Pointer

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

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.

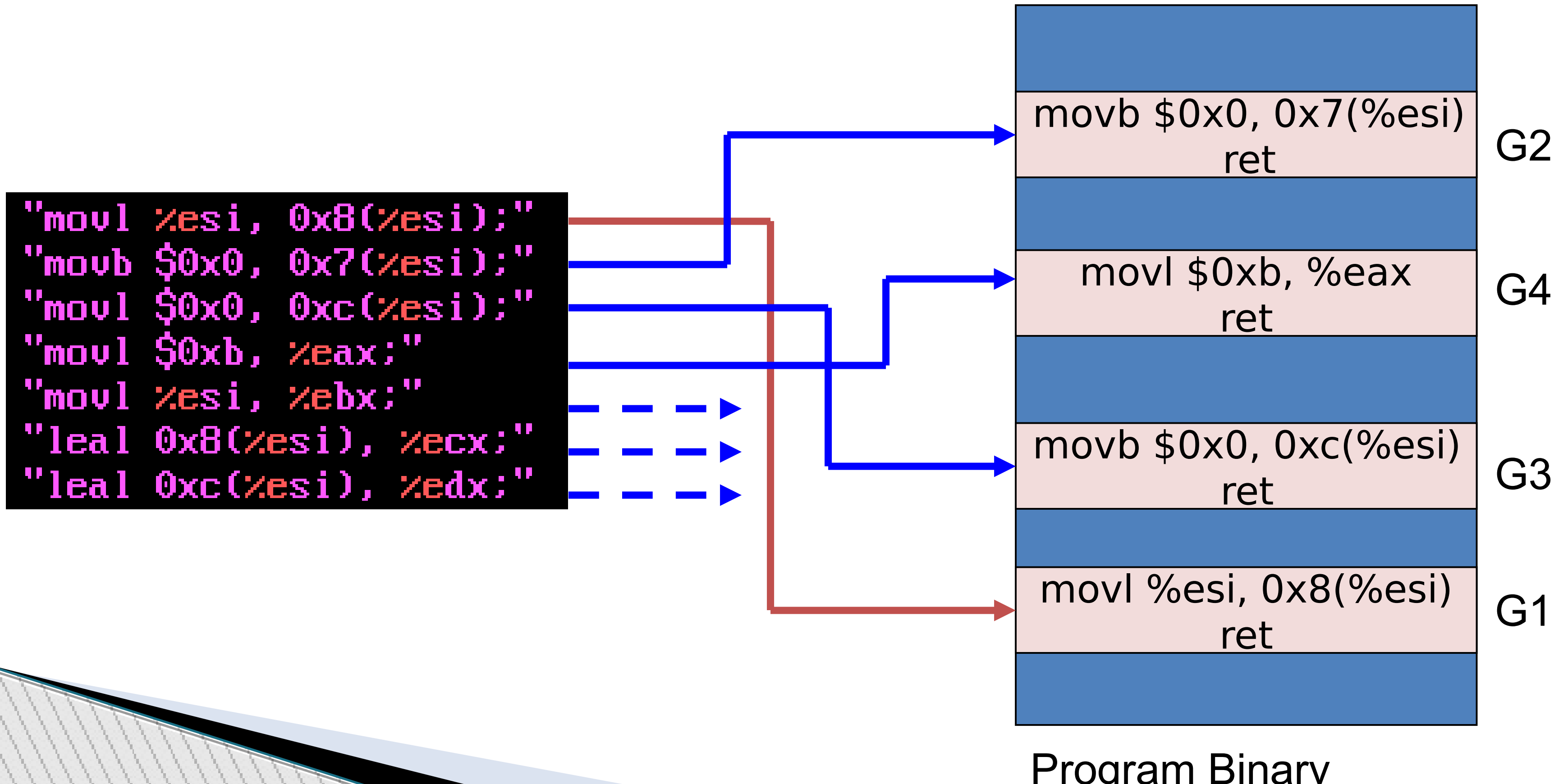
```
useful instruction(s)
ret
```

- Useful instructions : should not transfer control outside the gadget.

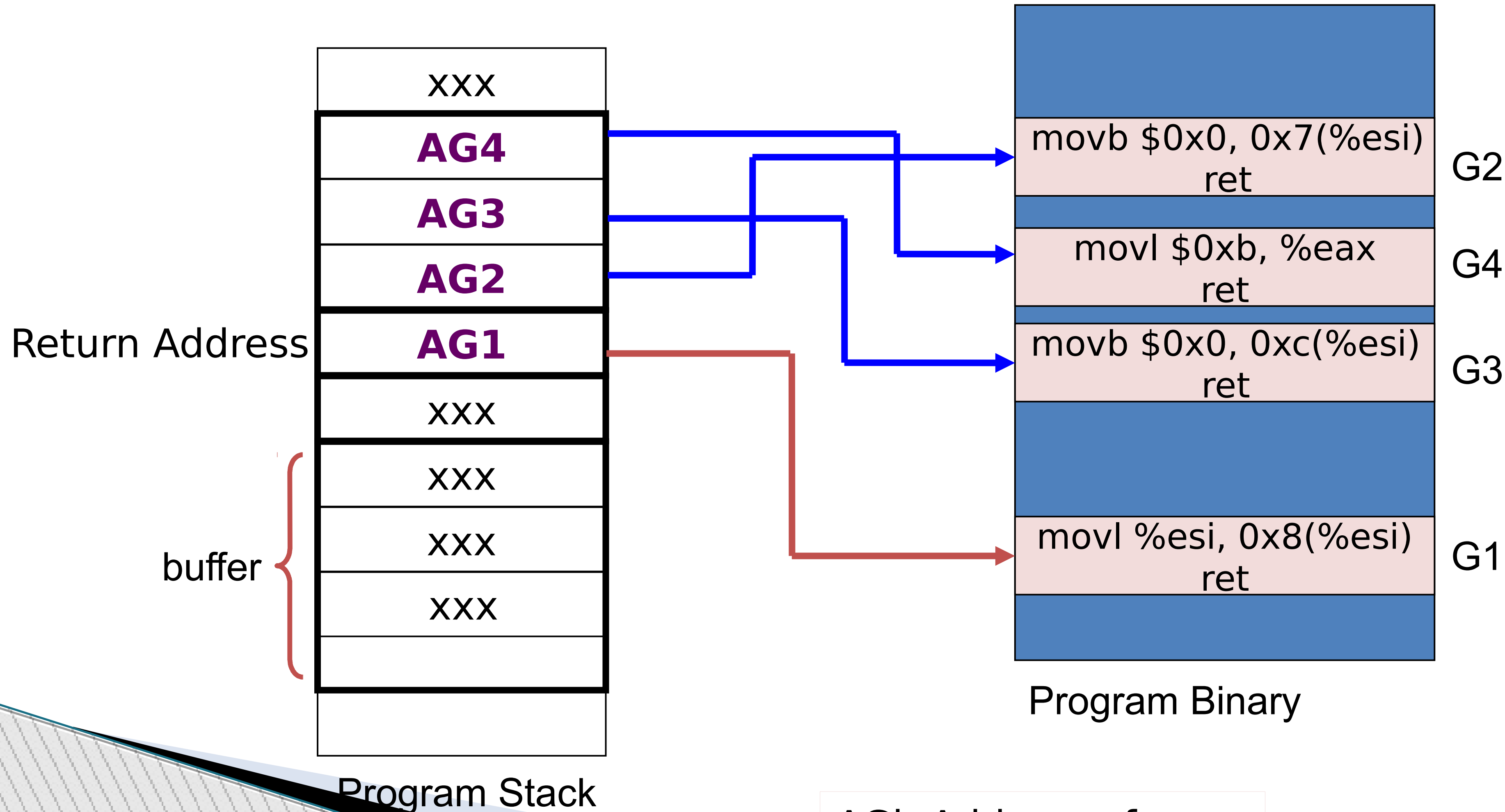
- This is a pre-processing step by statically analysing the libc library.

Step 2: Stitching

- Stitch gadgets so that the payload is built



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 intended

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

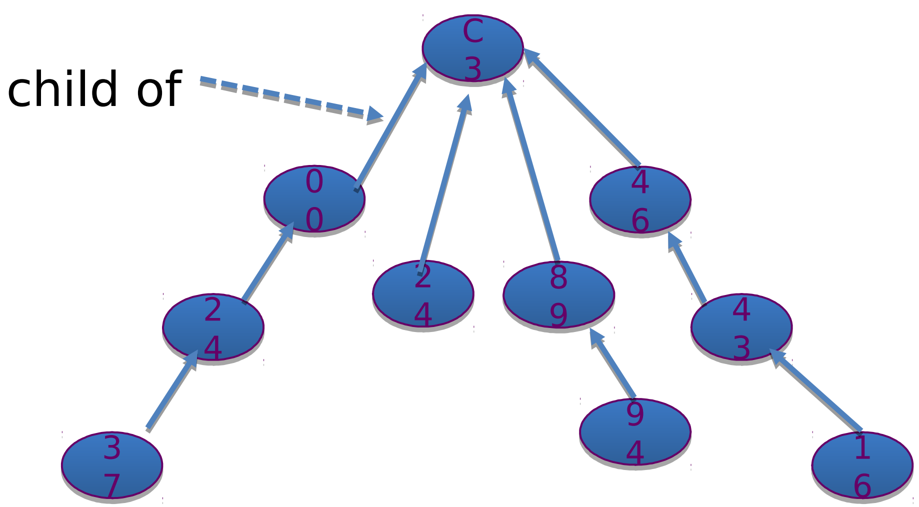
Highly likely to find many diverse instructions of this form in x86.
Not so likely to have such diverse instructions in RISC processors.

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 (RET instruction).
- Build a trie data structure with 0xc3 as a root.
- Every path from leaf to the root is a possible gadget.



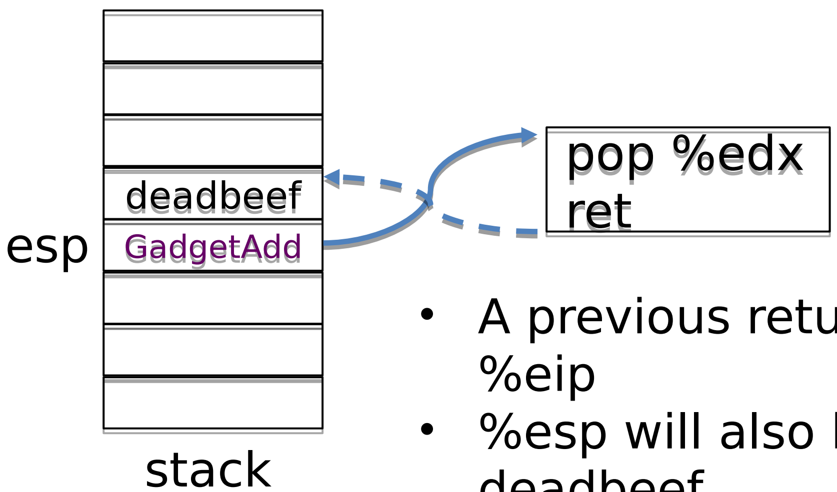
Finding Gadgets

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

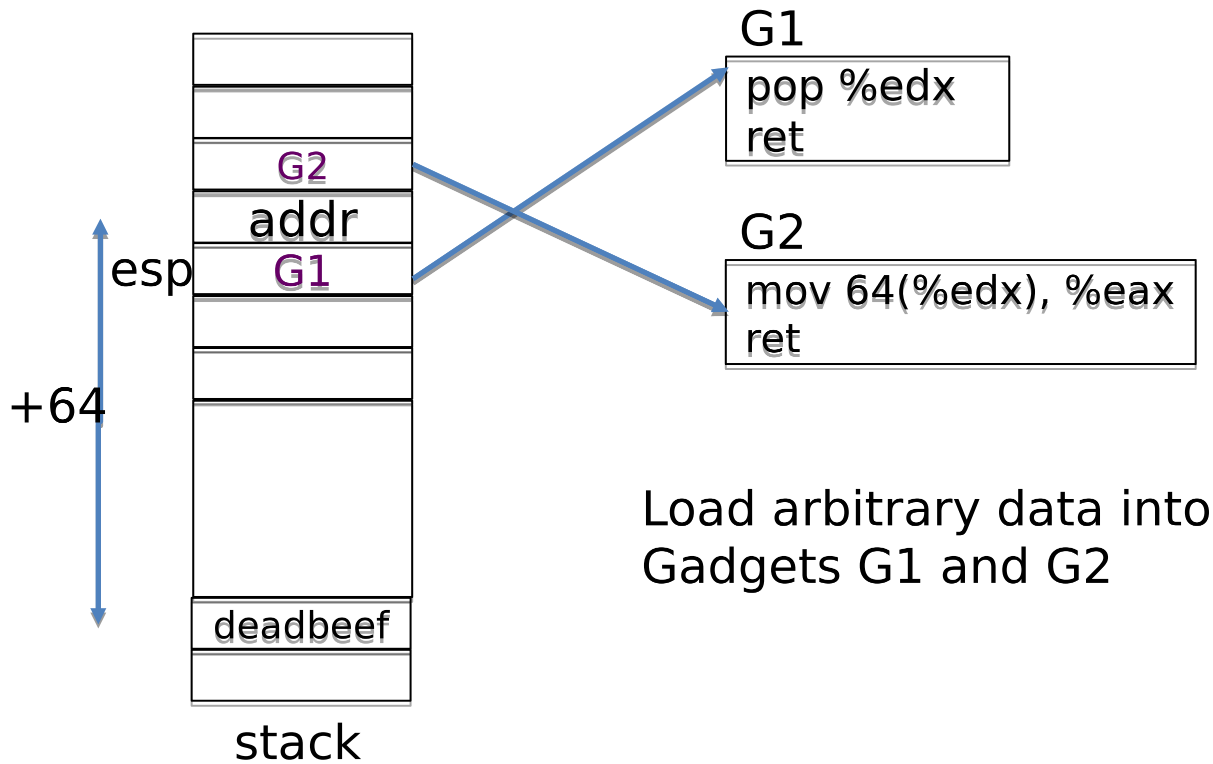
Gadgets : Constant into Register

Loading a constant into a register (edx = deadbeef)



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

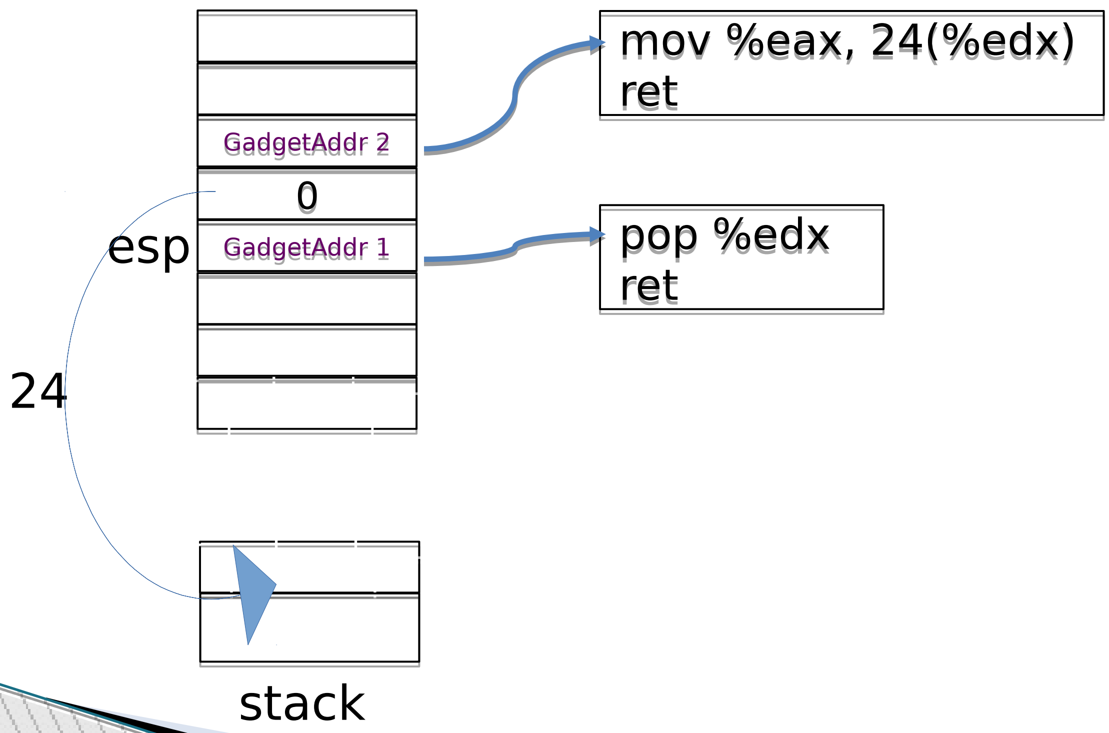
Gadgets : Arbitrary Data into eax



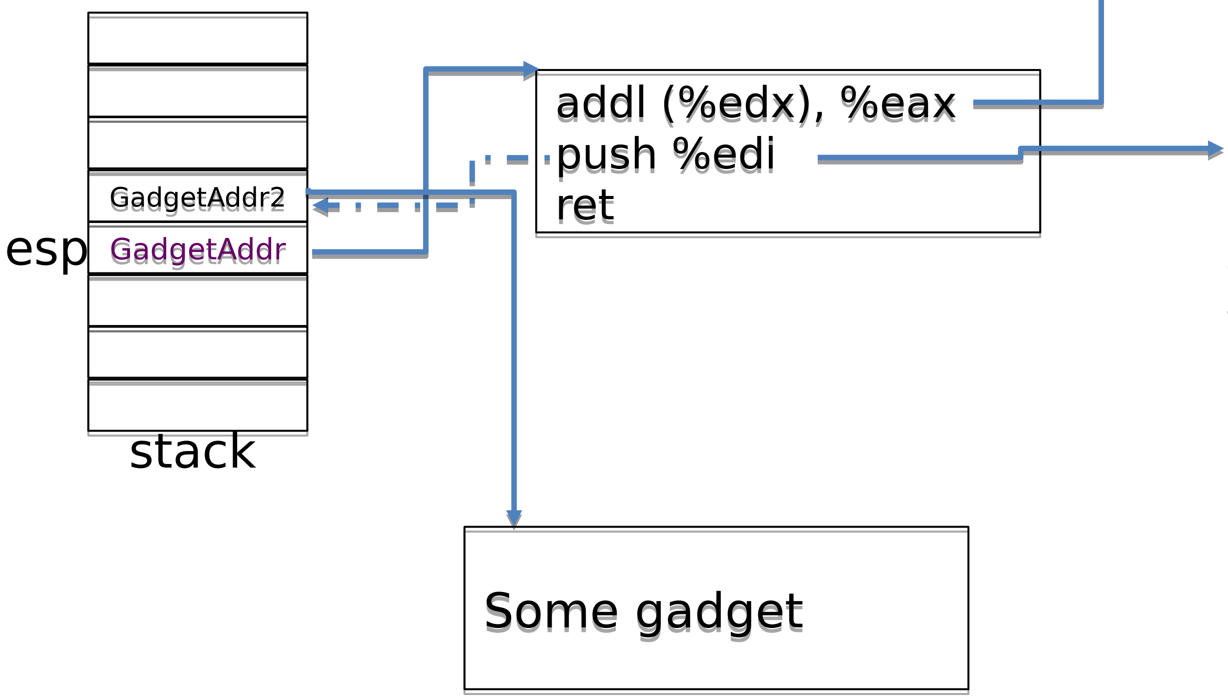
Load arbitrary data into %eax register using Gadgets G1 and G2

Gadgets: Store Constants

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



Gadget: Addition

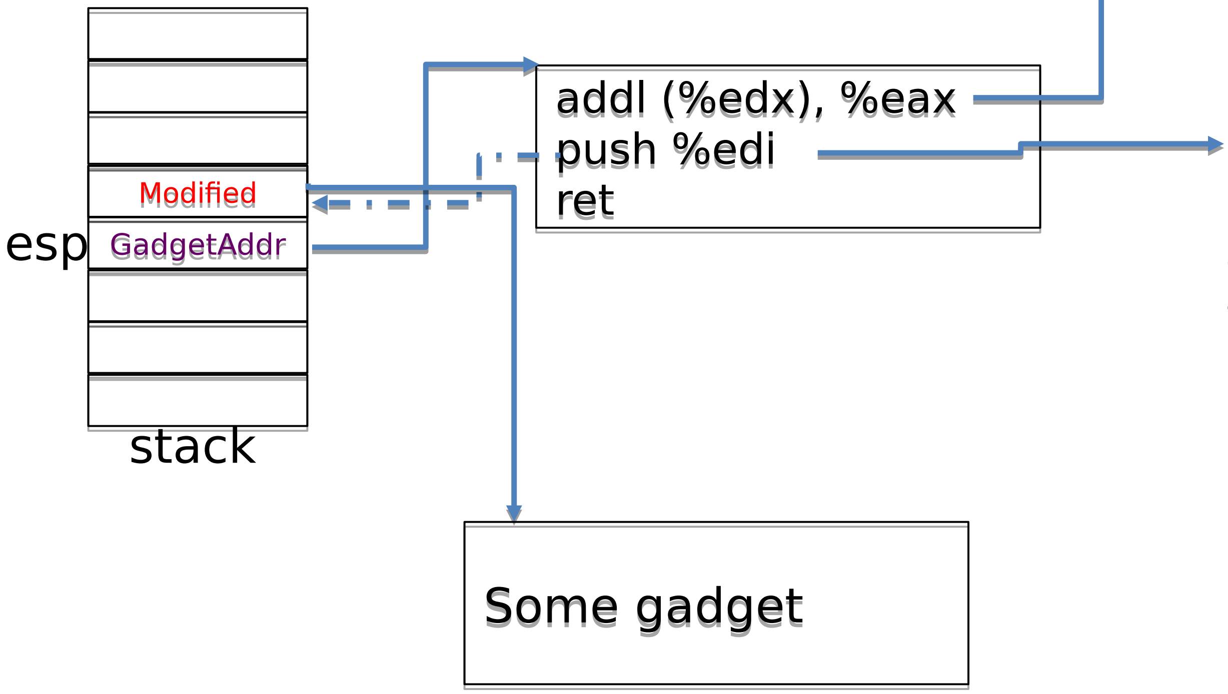


Add the memory pointed to by %edx to %eax.
The result is stored in %eax

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: Addition

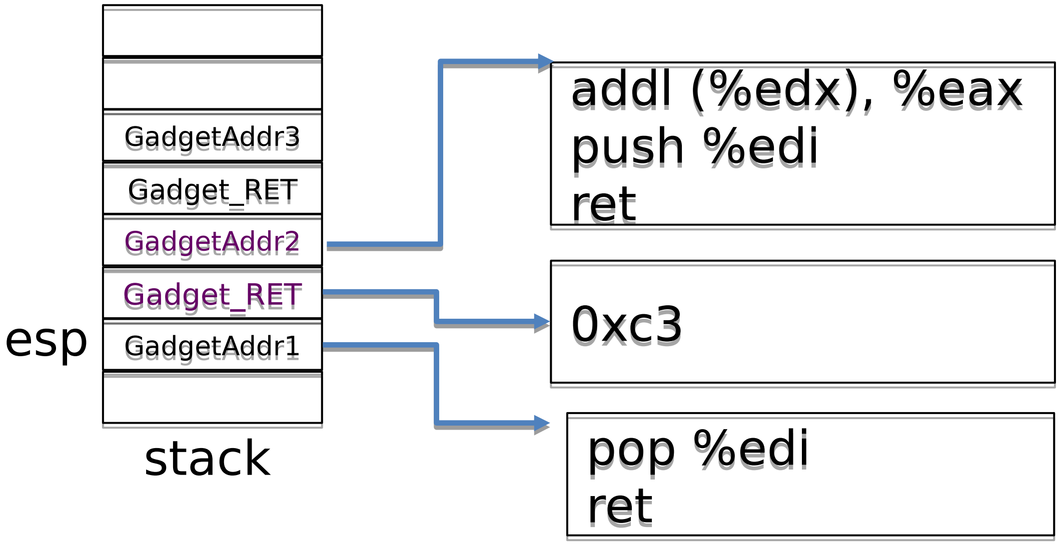


Add the memory pointed
to by %edx to %eax.
The result is stored in %eax

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!

Gadgets: Addition with NOP

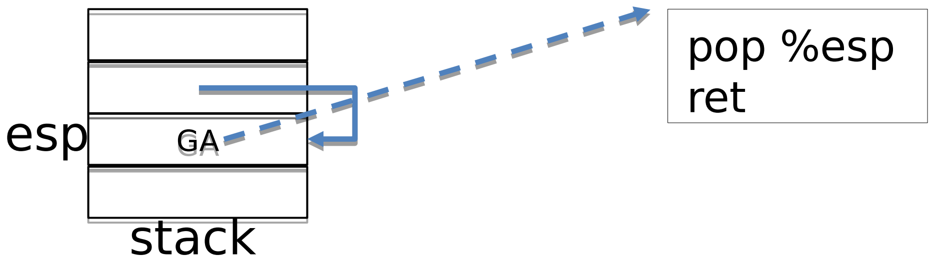


1. First put gadget ptr for 0xC3 into %edi
2. 0xC3 corresponds to NOP in ROP
3. push %edi in gadget 2 just pushes 0xc3 back into the stack Therefore not disturbing the stack contents
4. Gadget 3 executes as planned

0xc3 is ret in ROP and ret is equivalent to NOP instruction

Unconditional Branches

- Changing the %esp



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.

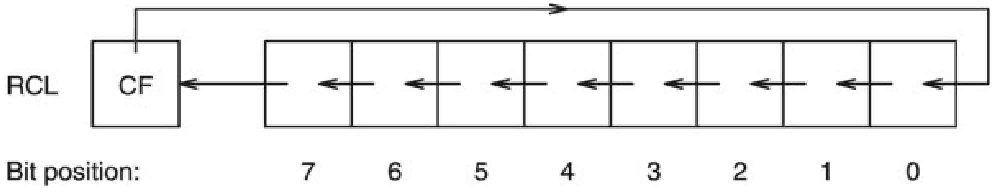
Step 2: Indirect way to transfer flags to memory

Several instructions operate using the contents of the flags

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

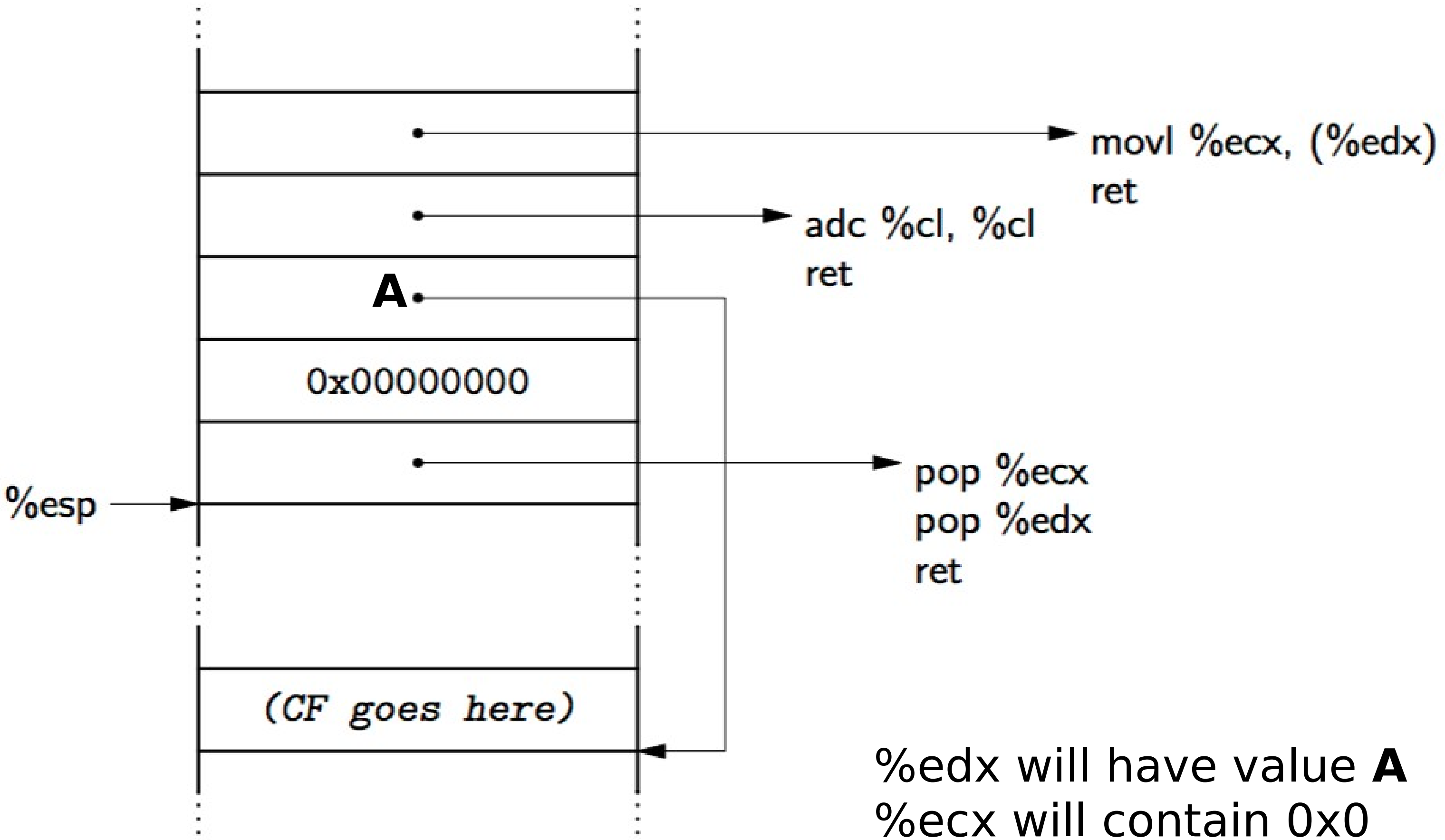
(if eax and ebx are 0 initially, then the result will be either 1 or 0 depending on the CF)

RCL : rotate left with carry.



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

Gadgets: 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.

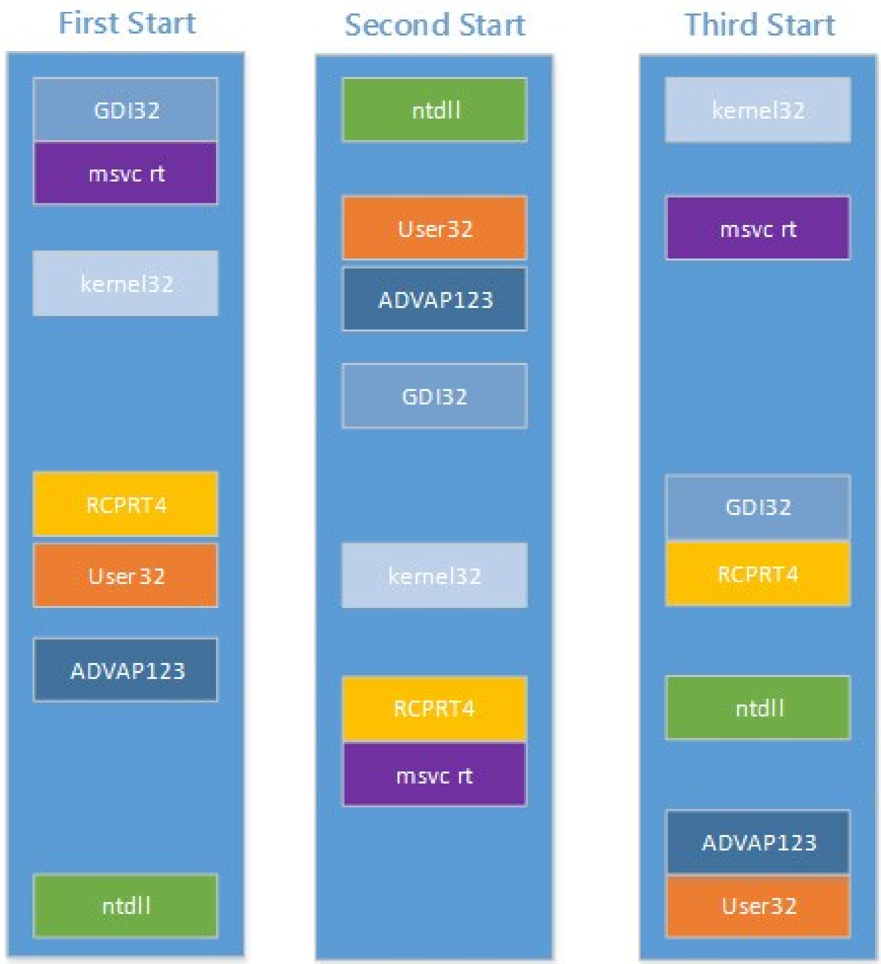
  Eg. ROPGadget (https://github.com/JonathanSalwan/ROPgadget)

  Ropper (https://github.com/sashs/Ropper)

Address Space Layout Randomization (ASLR)

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

```
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

```
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

Two requirements:-

- Make the code relocatable.
- Generate random address.

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;
}
```

```
make lib_reloc
gcc -g -c mylib.c -o mylib.o
gcc -shared -o libmylib.so mylib.o
```

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;
}
```

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

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

```
$ 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
```

Store binary value in the symbol memory location

Offset in memory wh[illegible]e the fix needs to be made

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_in
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.
```

```
entries:
rt__
erClasses
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.

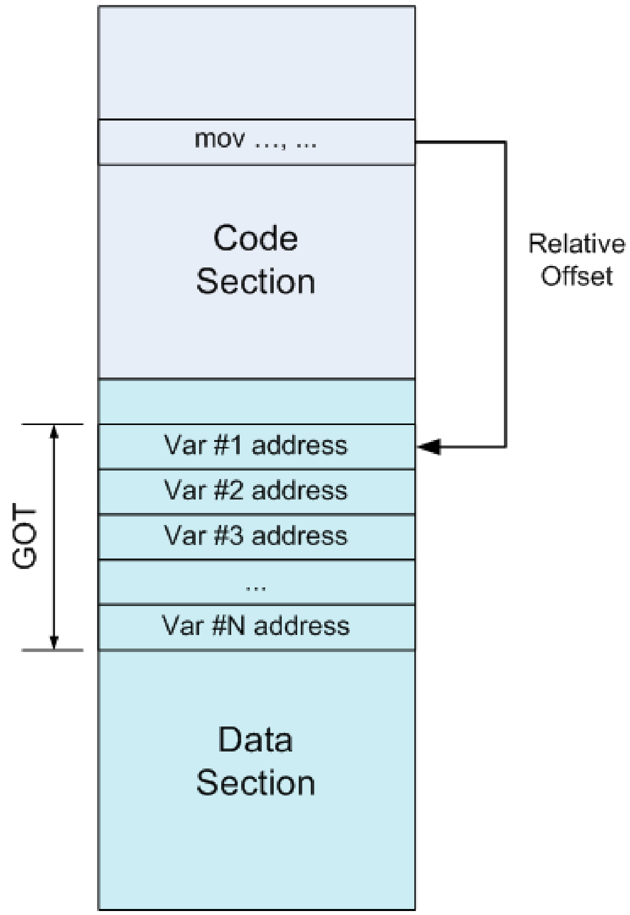
Position Independent Executable

- 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.

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
```

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.

Non Executable Stack Attack Prevention

```
        .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 (.data)

Text section (.text)

The macro for the GOT is known by the linker. %ecx will now contain the offset to GOT (addl $_GLOBAL_OFFSET_TABLE_, %ecx)

Load the absolute address of myglob from the GOT into %eax (movl myglob@GOT(%ecx), %eax)

Fills %ecx with the eip of the next instruction. (__i686.get_pc_thunk.cx)
Why do we need this indirect way of doing this?
In this case what will %ecx contain?

More

```
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!

Internals: 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

Deep Within the Kernel (randomizing the data section)

```
static struct keydata {
        __u32 count; /* already shifted to the final position */
        __u32 secret[12];
} ____cacheline_aligned ip_keydata[2];
  ...
/*
 * Get a random word for internal kernel use only. Similar to urandom but
 * with the goal of minimal entropy pool depletion. As a result, the random
 * value is not cryptographically secure but for several uses the cost of
 * depleting entropy is too high
 */
DEFINE_PER_CPU(__u32 [4], get_random_int_hash);
unsigned int get_random_int(void)
{
        struct keydata *keyptr;
        __u32 *hash = get_cpu_var(get_random_int_hash);
        int ret;

        keyptr = get_keyptr();
        hash[0] += current->pid + jiffies + get_cycles();

        ret = half_md4_transform(hash, keyptr->secret);
        put_cpu_var(get_random_int_hash);

        return ret;
}
```

Deep Within the Kernel (randomizing the data section)

```
hash[0] += current->pid + jiffies + get_cycles()
```

- The address of the first element of the 'hash[0]' array.
- The currently executing process ID for the processor that handles this.
- The system's jiffies value.
- CPU cycles number.

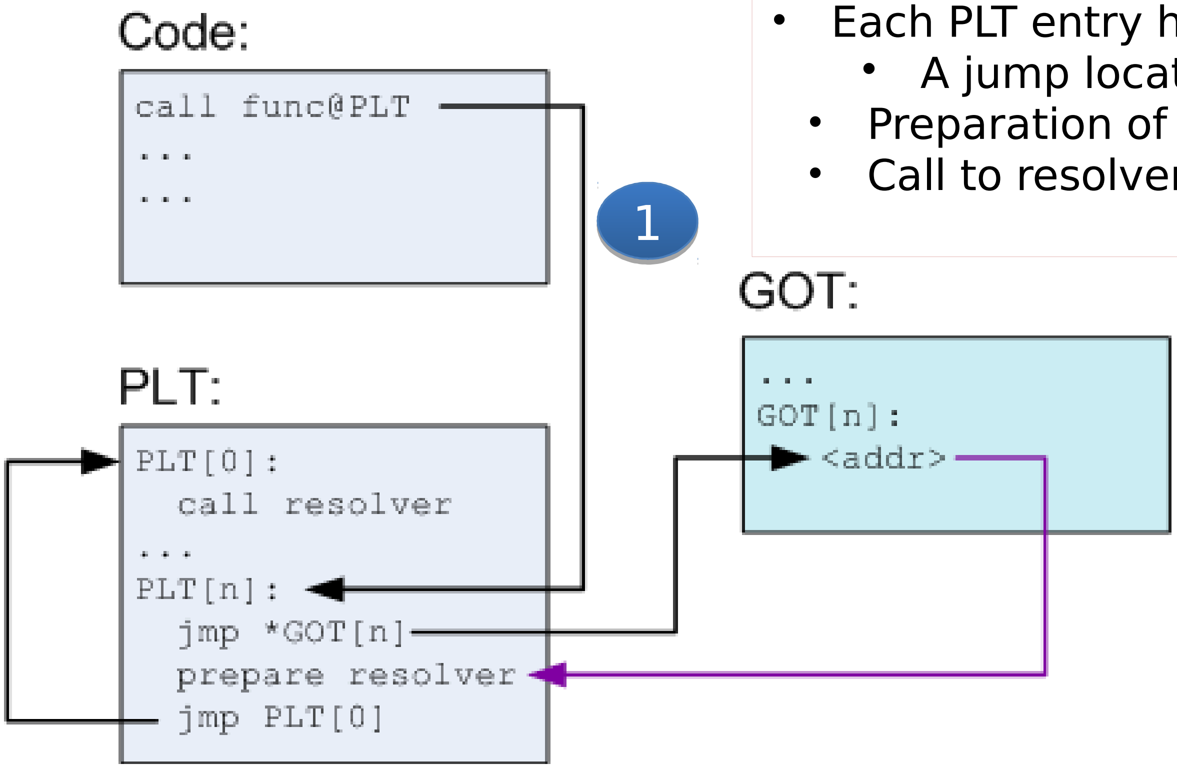
```
static inline cycles_t get_cycles(void)
{
        unsigned long long ret = 0;

#ifndef CONFIG_X86_TSC
        if (!cpu_has_tsc)
                return 0;
#endif
        rdtscll(ret);

        return ret;
}
```

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

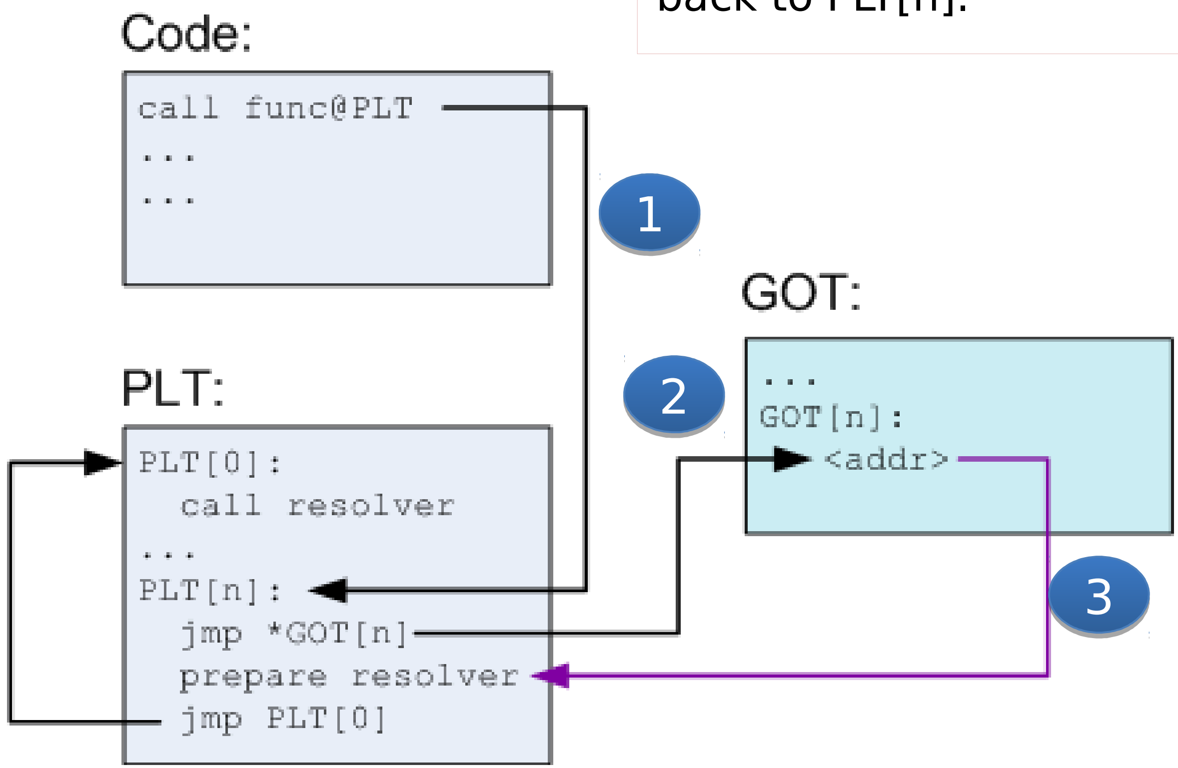
- Instead of directly calling func, invoke an offset in the PLT instead.
- PLT is part of the executable text section, and consists of one entry for each external function the shared library calls.
- Each PLT entry has
  - A jump location to a specific GOT entry
  - Preparation of arguments for a ‘resolver’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



First Invocation of Func

First Invocation of func

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

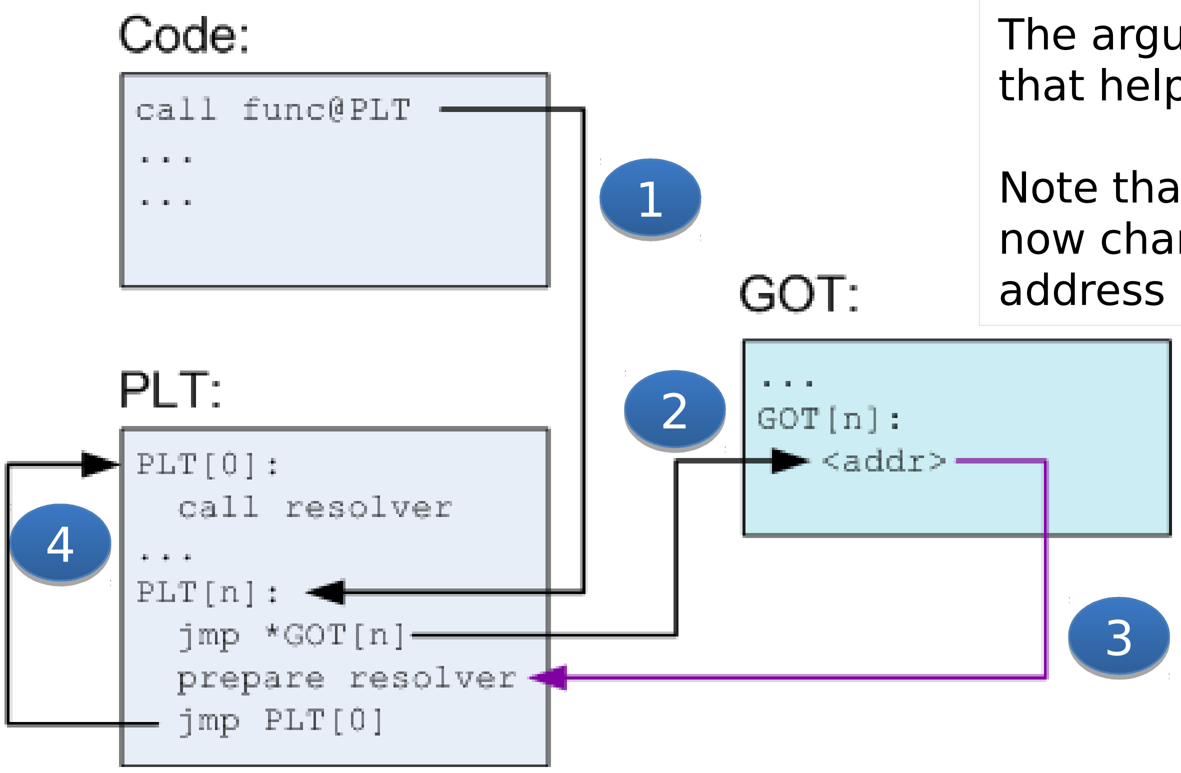


First Invocation of Func

(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;
}
```

```
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)

```
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

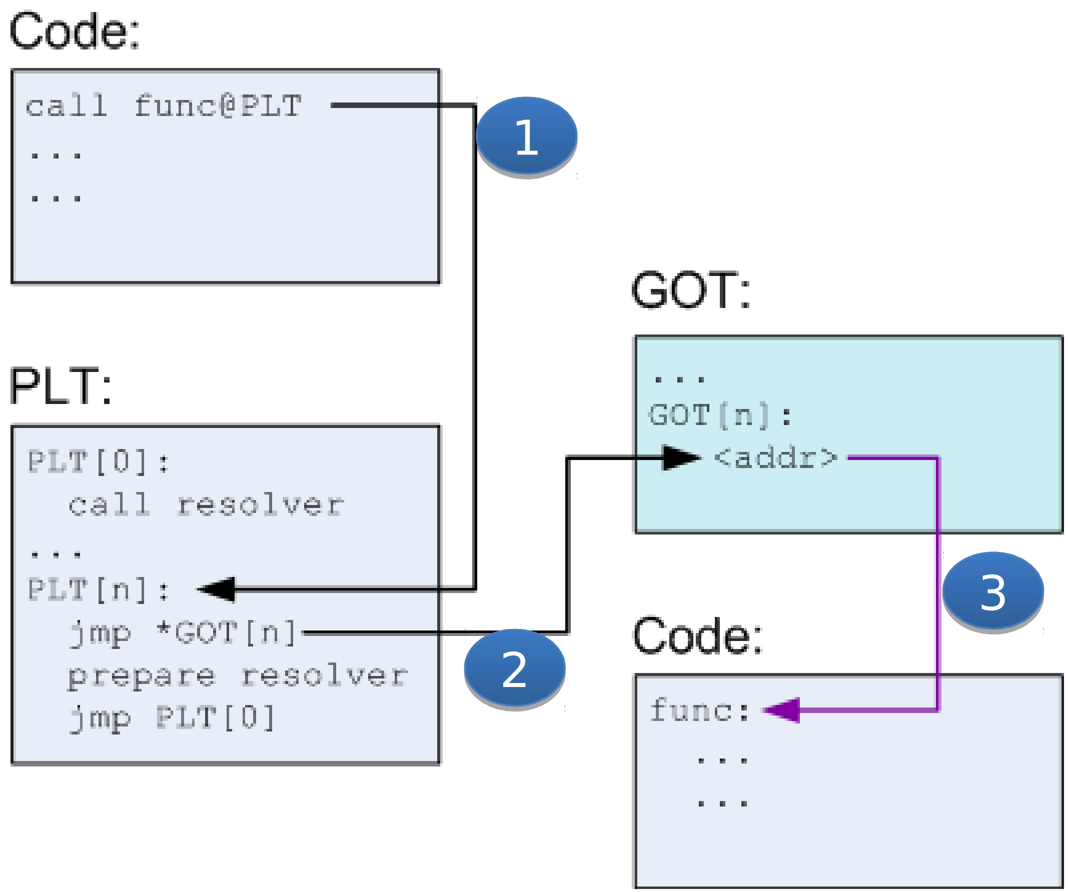
```
Disassembly of section .plt:

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.

Bypassing ASLR

- Brute force.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

Bypassing ASLR

- Brute force.

```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#      ← Got the root shell!
```

That's for the classes