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 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)

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

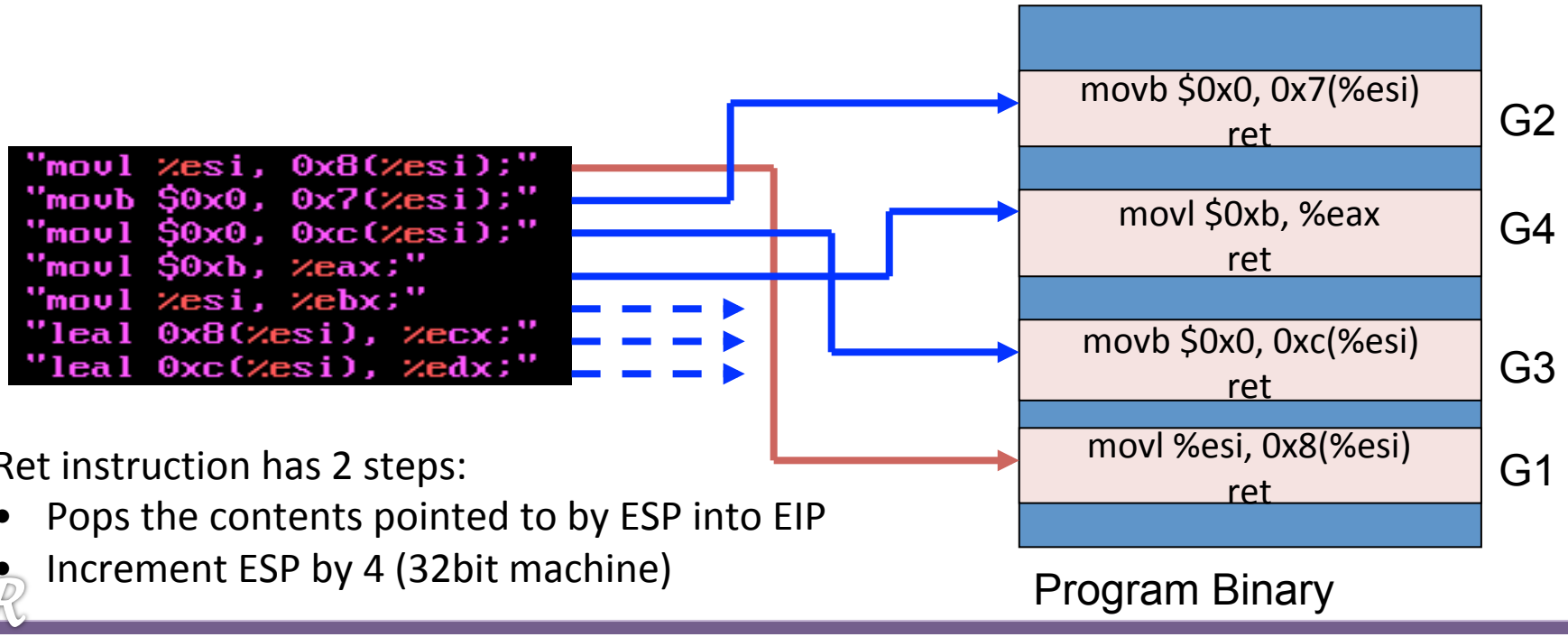
```
useful instruction(s)
ret
```

- Useful instructions : should not transfer control outside the gadget

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

Step 2: Stitching

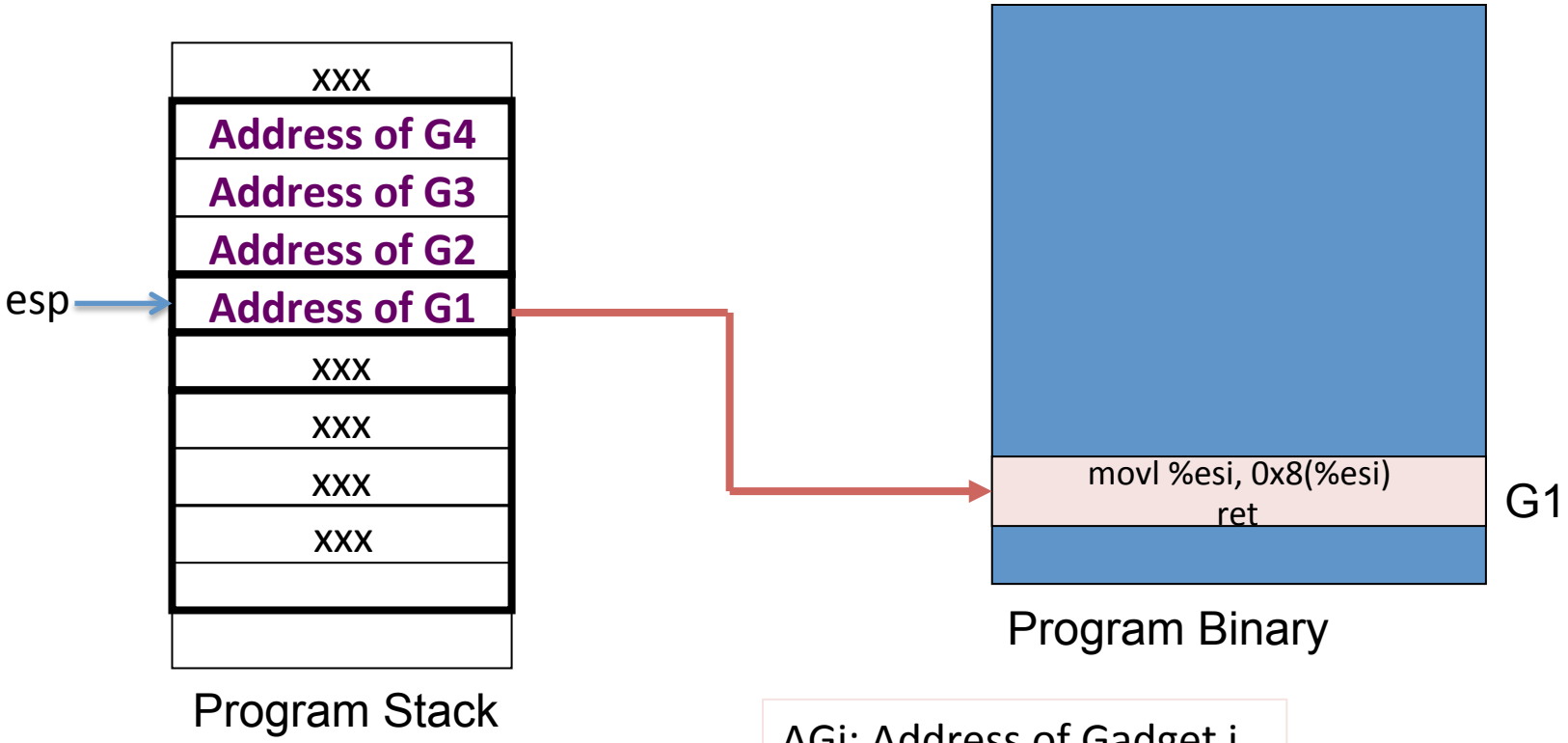
- Stitch gadgets so that the payload is built



Ret instruction has 2 steps:

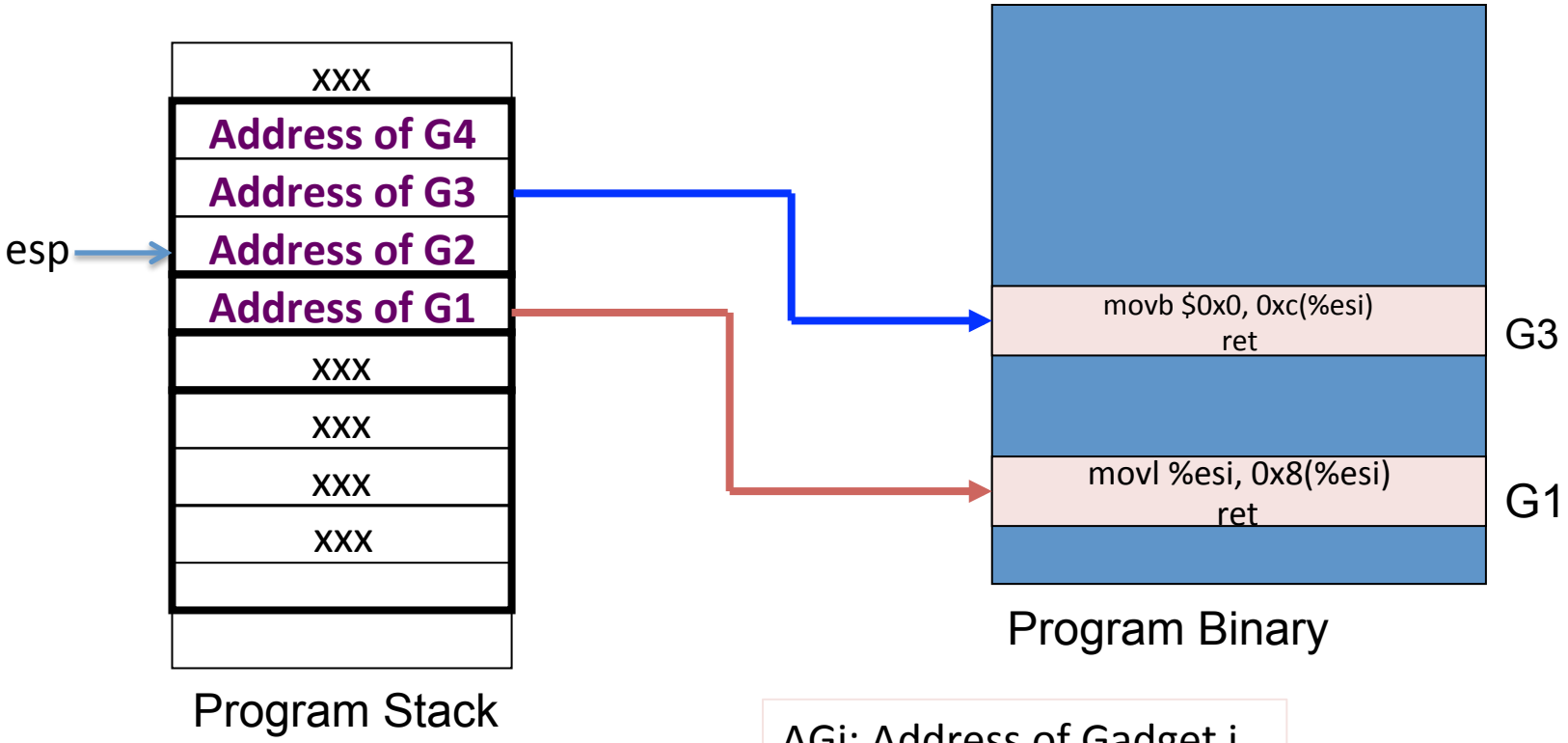
- Pops the contents pointed to by ESP into EIP
- Increment ESP by 4 (32bit machine)

Step 3: Construct the Stack



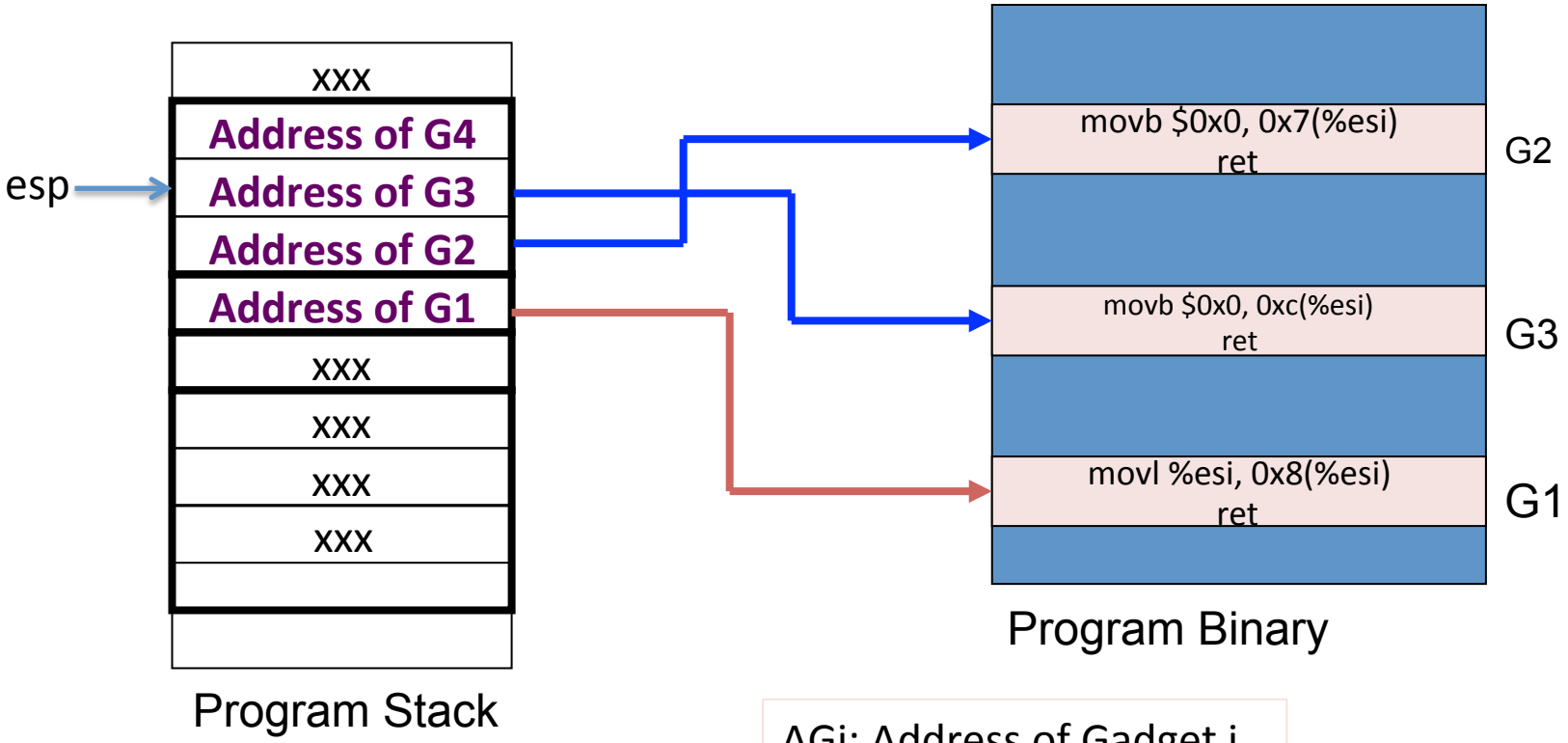
AGi: Address of Gadget i

Step 3: Construct the Stack



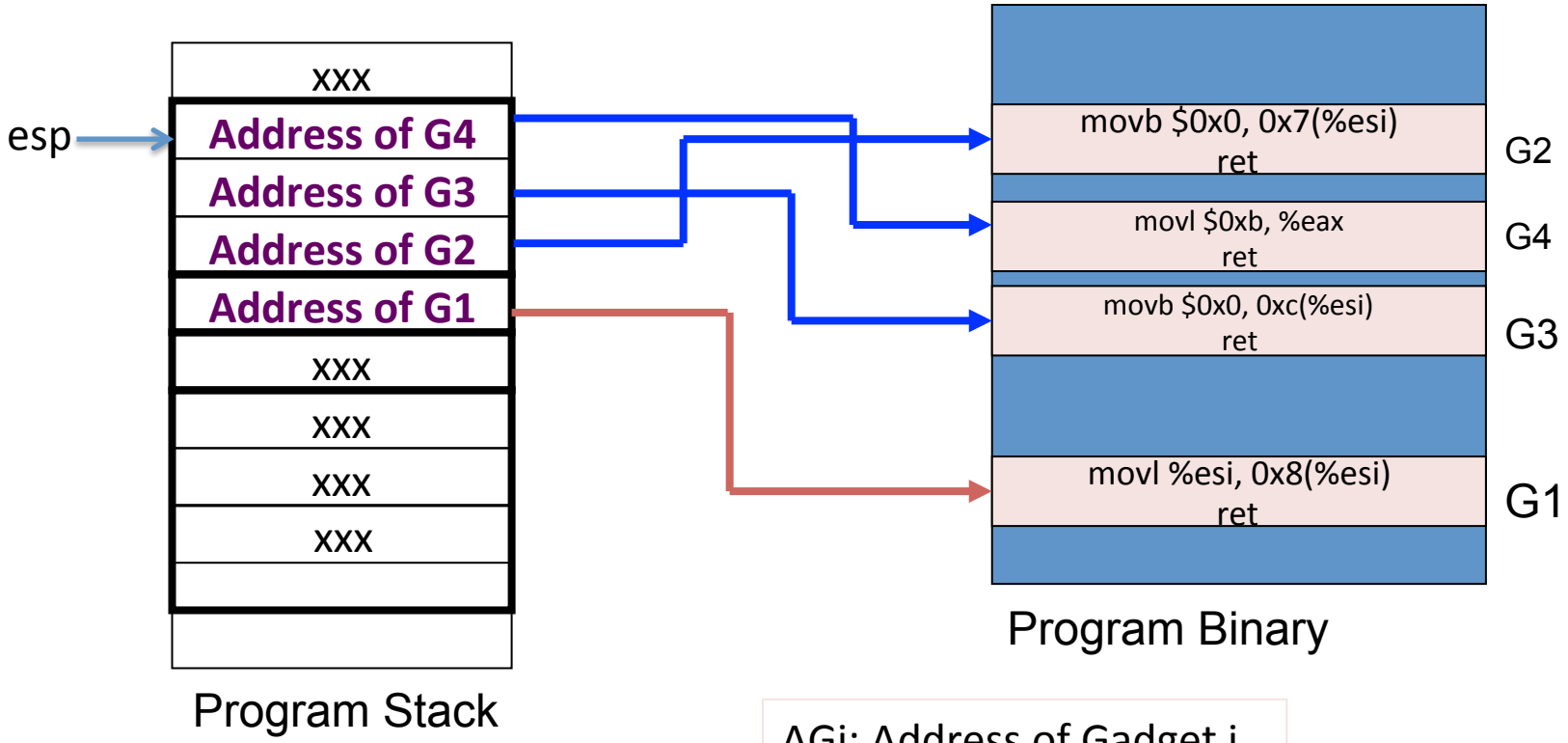
AGi: Address of Gadget i

Step 3: Construct the Stack



AGi: Address of Gadget i

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..

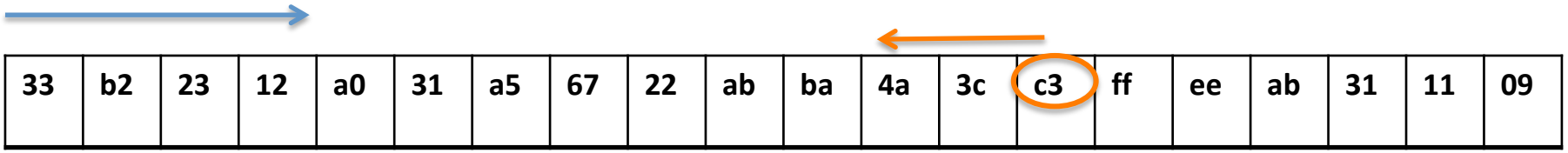
```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

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

Finding Gadgets

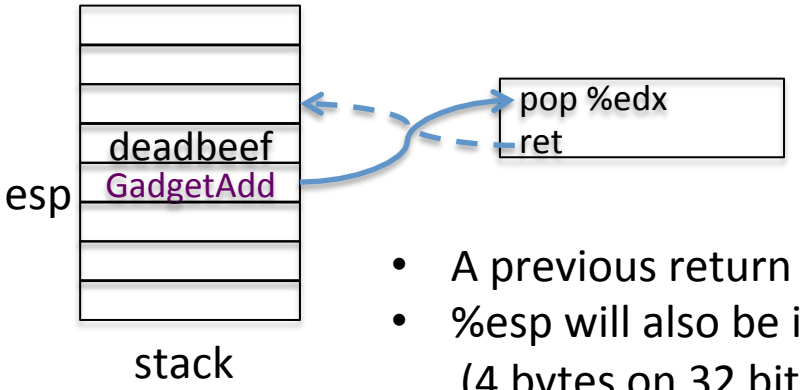


- Scan libc from the beginning toward the end
- If 0xc3 is found
  - Start scanning backward
  - With each byte, ask the question if the subsequence forms a valid instruction
  - If yes, add as child
  - If no, go backwards until we reach the maximum instruction length (20 bytes)
  - Repeat this till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

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

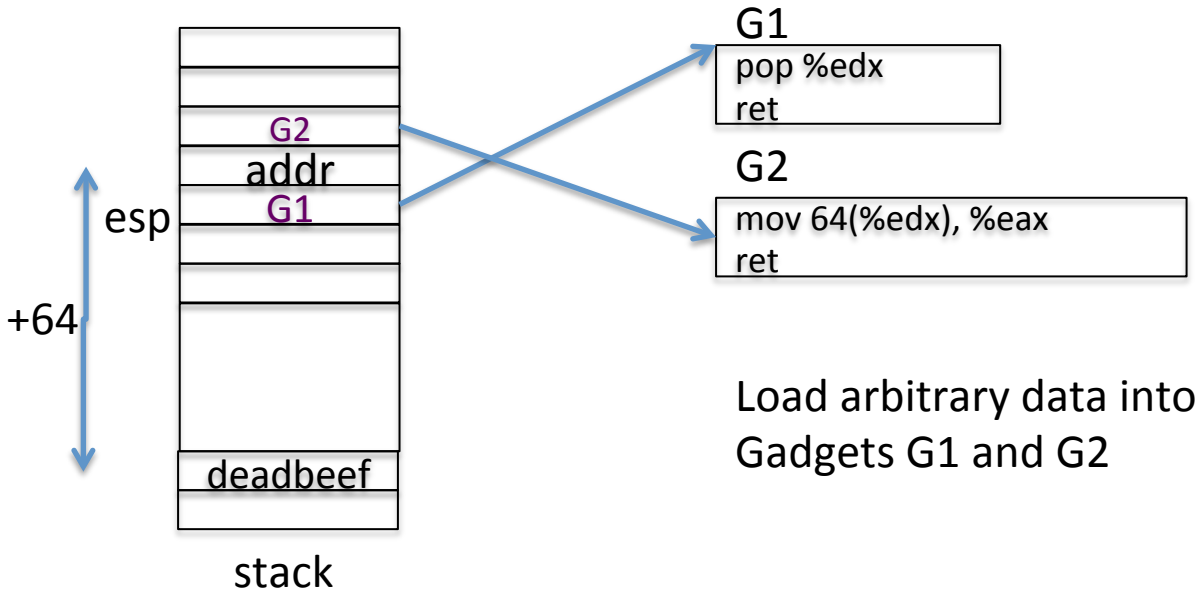
More about Gadgets

- Example Gadgets
  - 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

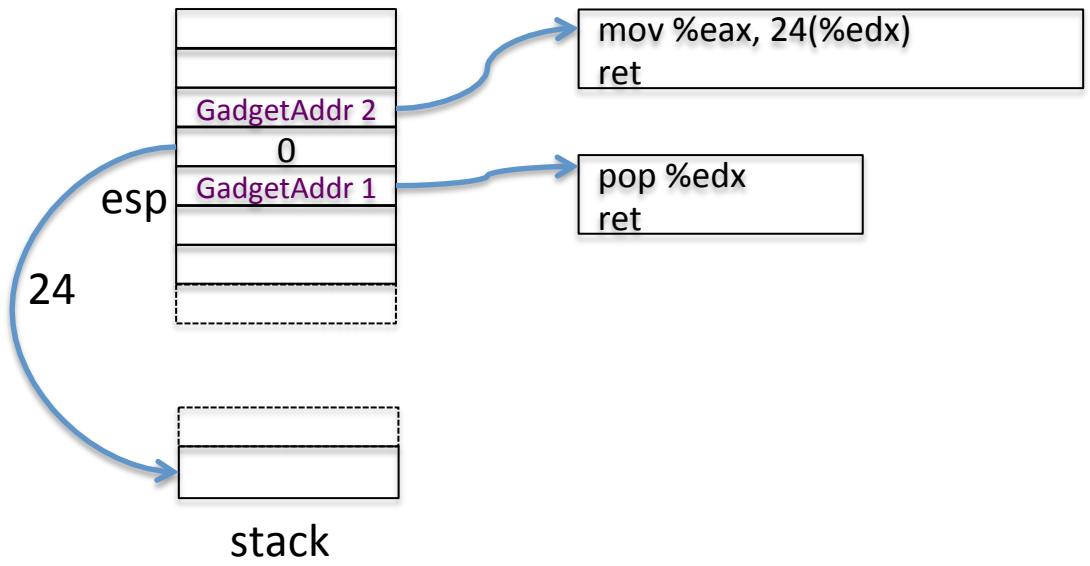
Stitching Gadgets



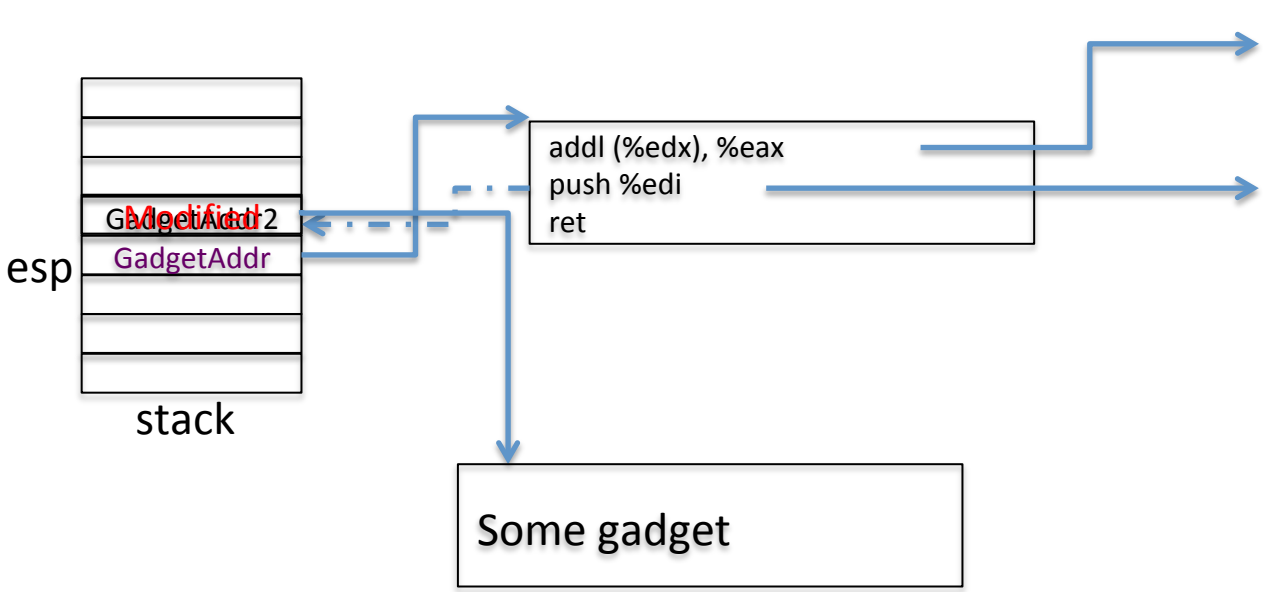
Load arbitrary data into %eax register using Gadgets G1 and G2

Store Gadget

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



Gadget for addition

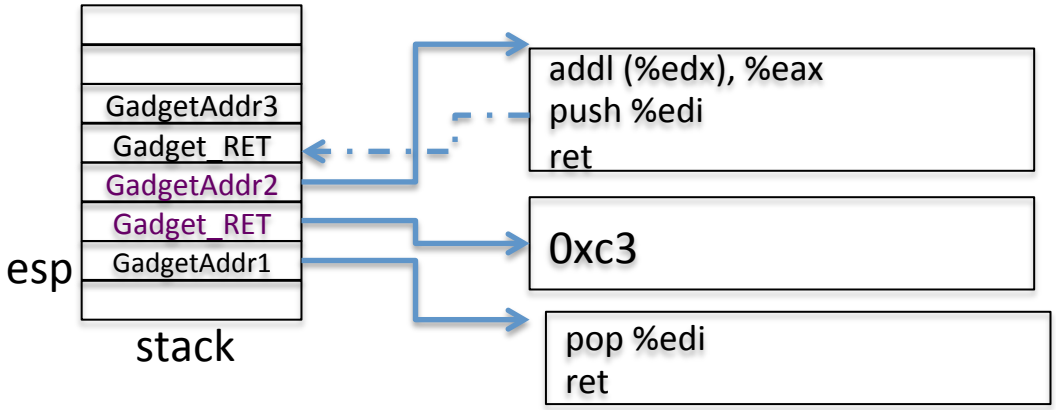


Add the memory pointed to by %edx to %eax.
The result is stored in %eax

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)

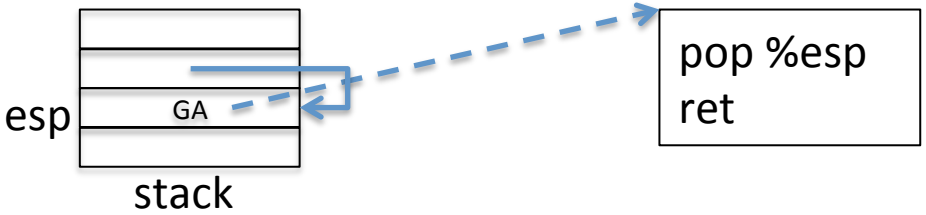


1. First put gadget ptr for 0xC3 into %edi
2. 0xC3 corresponds to NOP in ROP
3. push %edi in gadget 2 just pushes 0xc3 back into the stack
   Therefore not disturbing the stack contents
4. Gadget 3 executes as planned

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

Unconditional Branch in ROP

- Changing the %esp causes unconditional jumps



Tools

- 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)