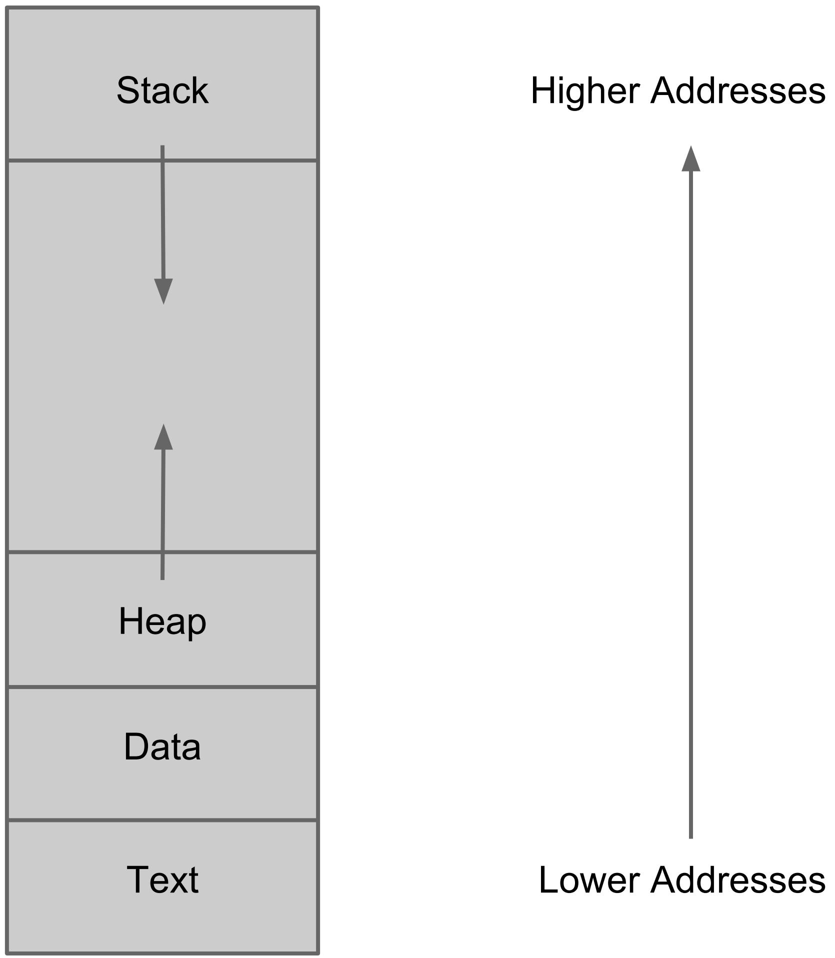
Buffer Overflow Attacks

IA32 Linux
Virtual Address Space



Stack

Heap

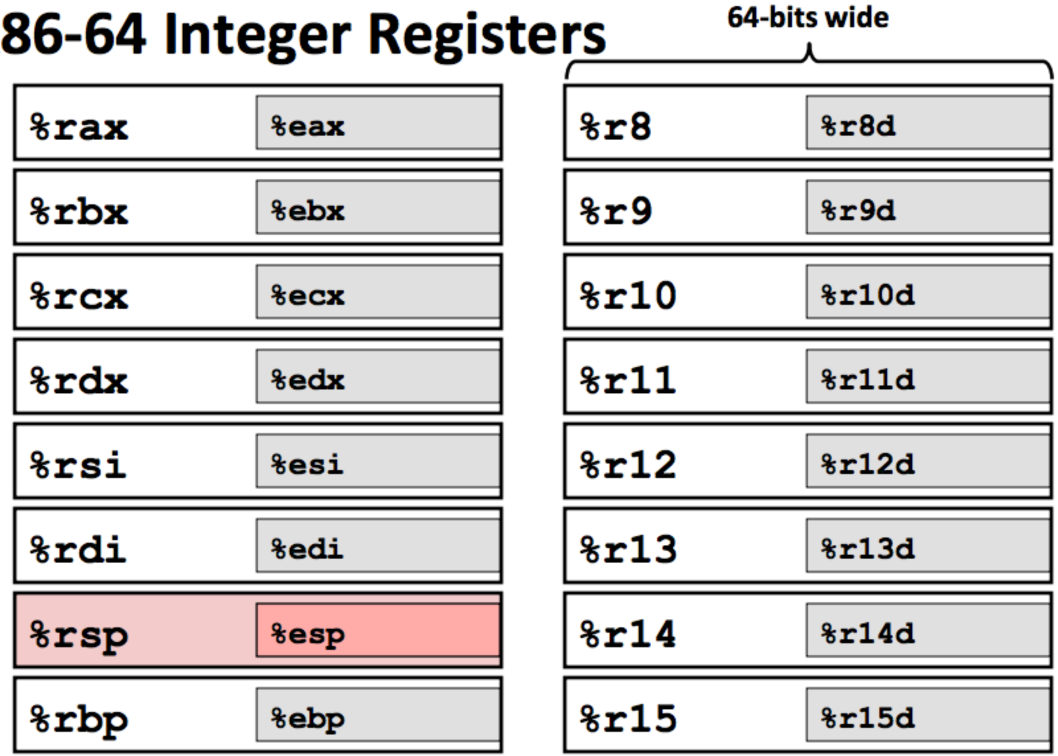
Data

Text

Higher Addresses

Lower Addresses

x86-64 Integer Registers

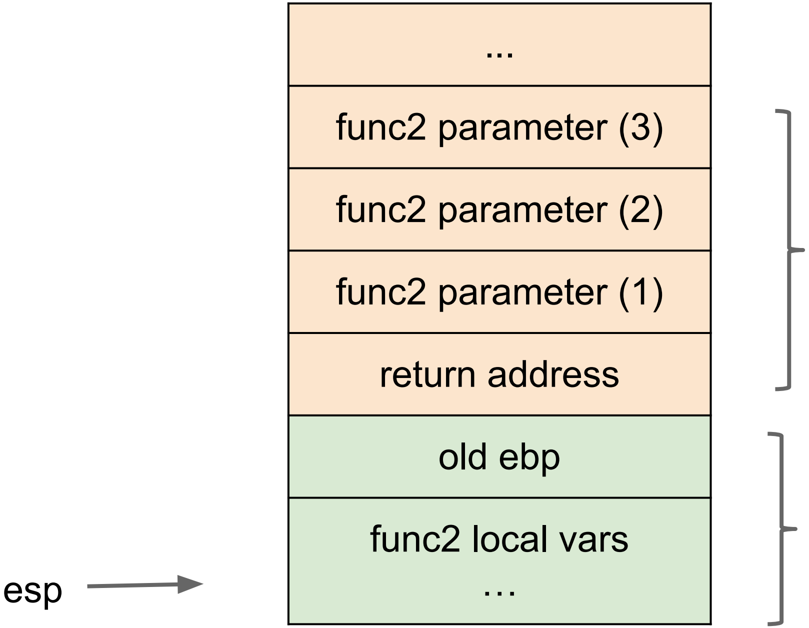


`%rax`	`%eax`	`%r8`	`%r8d`
`%rbx`	`%ebx`	`%r9`	`%r9d`
`%rcx`	`%ecx`	`%r10`	`%r10d`
`%rdx`	`%edx`	`%r11`	`%r11d`
`%rsi`	`%esi`	`%r12`	`%r12d`
`%rdi`	`%edi`	`%r13`	`%r13d`
`%rsp`	`%esp`	`%r14`	`%r14d`
`%rbp`	`%ebp`	`%r15`	`%r15d`

- Extend existing registers, and add 8 new ones; *all* accessible as 8, 16, 32, 64 bits.

Stack and Base Pointers

- Stack is made up of stack frames
- Stack frames contain:
    - parameters, local variables, return addresses, instruction pointer
- Stack Pointer: points to the top of the stack (lowest address)
- Frame Pointer: Points to the base of the frame



...

func2 parameter (3)

func2 parameter (2)

func2 parameter (1)

return address

old ebp

func2 local vars
…

esp

caller_func stack frame

func2 stack frame

```
void caller_func() {
        func2( 1, 2, 3);
}

int func2( 1, 2, 3) {
        …
}
```

All content from these slides, including all code examples and attack examples come straight from “Low-Level Software Security by Example” by Ulfar Erlingsson, Yves Younan, and Frank Piessens.

Great paper! Go read it!

Attack 1: Stack-based Buffer Overflow

Clobber the return address!

Review from Tuesday

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

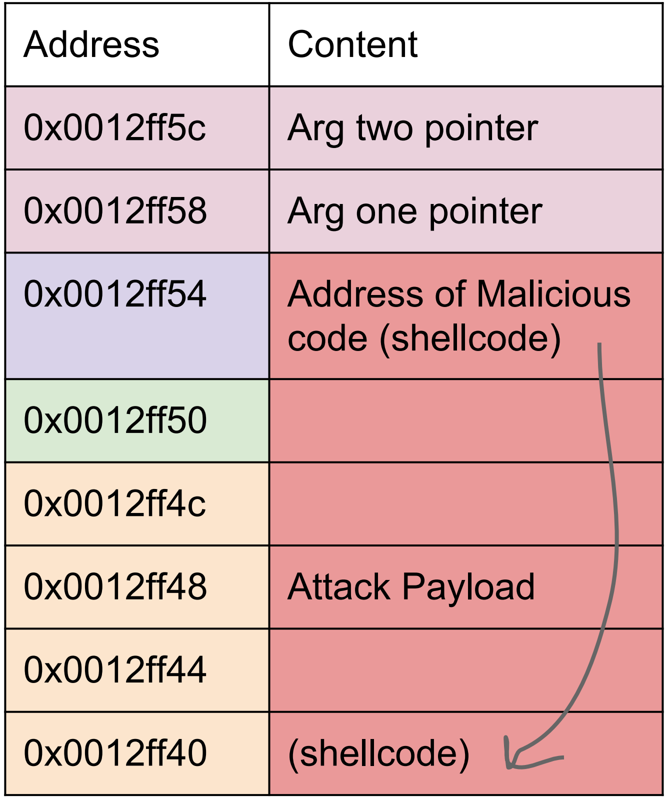
Address	Content
0x0012ff5c	Arg two pointer
0x0012ff58	Arg one pointer
0x0012ff54	Return Address
0x0012ff50	Saved Base Pointer
0x0012ff4c	Tmp Array (end)
0x0012ff48	
0x0012ff44	
0x0012ff40	Tmp Array (start)

Corrupted!

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

Address	Content
0x0012ff5c	Arg two pointer
0x0012ff58	Arg one pointer
0x0012ff54	Address of Malicious code (shellcode)
0x0012ff50	
0x0012ff4c	
0x0012ff48	Attack Payload
0x0012ff44	
0x0012ff40	

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```



Address	Content
0x0012ff5c	Arg two pointer
0x0012ff58	Arg one pointer
0x0012ff54	Address of Malicious code (shellcode)
0x0012ff50	
0x0012ff4c	
0x0012ff48	Attack Payload
0x0012ff44	
0x0012ff40	(shellcode)

Attack 1: Stack-based Buffer Overflow

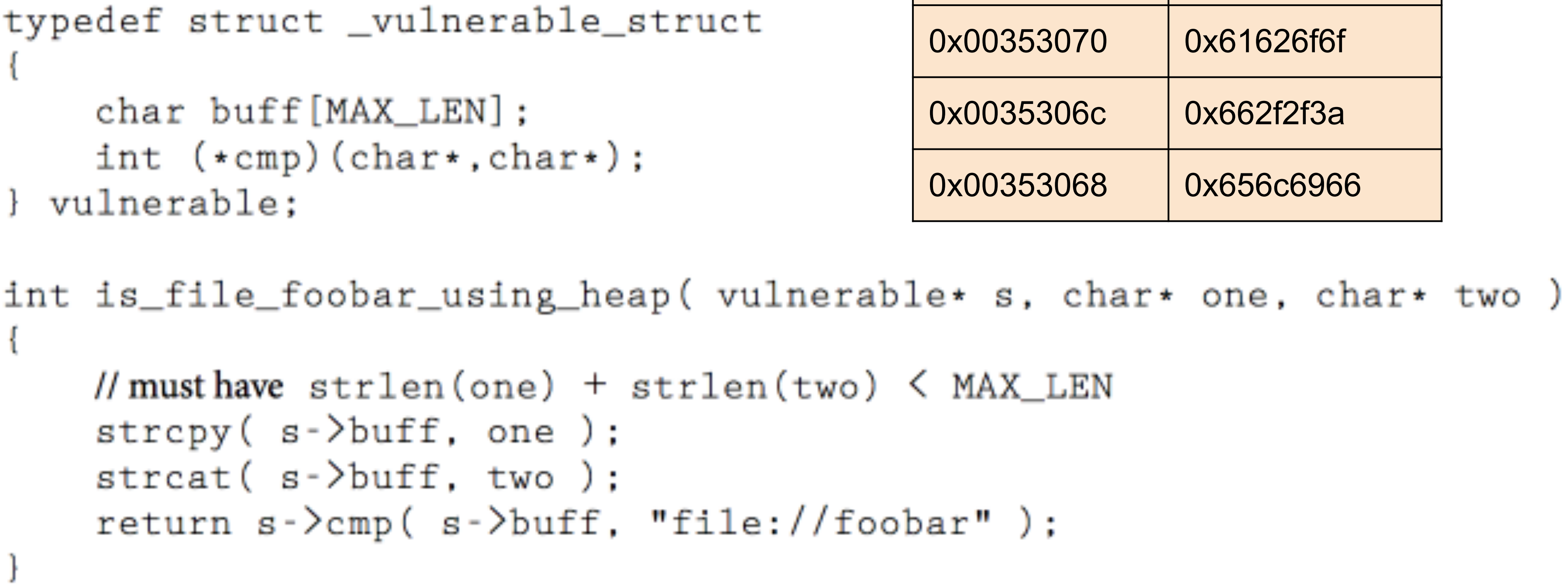
Caveats:

- Only addresses above buffer are changed
- What would happen if the attack payload contained null bytes or zeros?
- What if we corrupt %ebp instead of the return address?

Attack 2: Heap-based Buffer Overflows

Very similar to stack-based buffer overflow attacks except it affects data on the heap

Address	Content
0x00353078	0x004013ce
0x00353074	0x00000072
0x00353070	0x61626f6f
0x0035306c	0x662f2f3a
0x00353068	0x656c6966



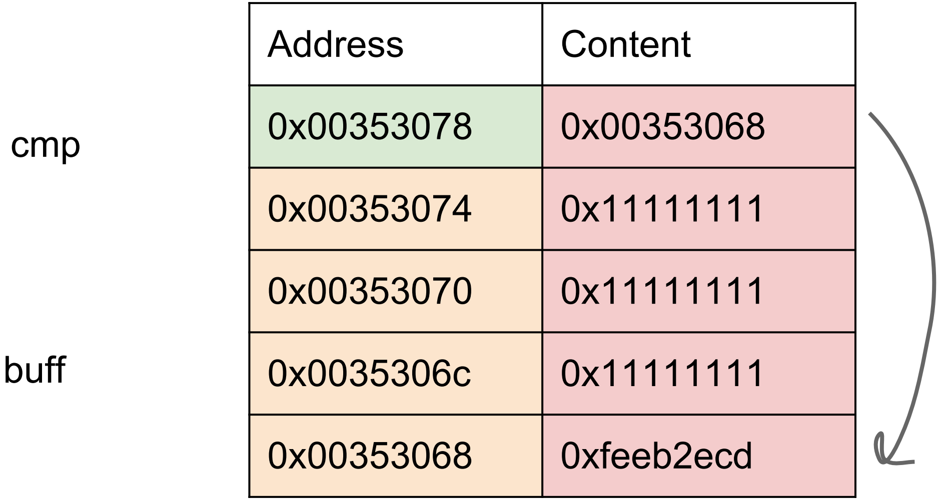
```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

	Address	Content
cmp	0x00353078	0x004013ce
	0x00353074	0x00000072
	0x00353070	0x61626f6f
buff	0x0035306c	0x662f2f3a
	0x00353068	0x656c6966

Translated
pointer to strcmp function
‘\0’ ‘\0’ ‘\0’ ‘r’
‘a’ ‘b’ ‘o’ ‘o’
‘f’ ‘/’ ‘/’ ‘:’
‘e’ ‘l’ ‘i’ ‘f’

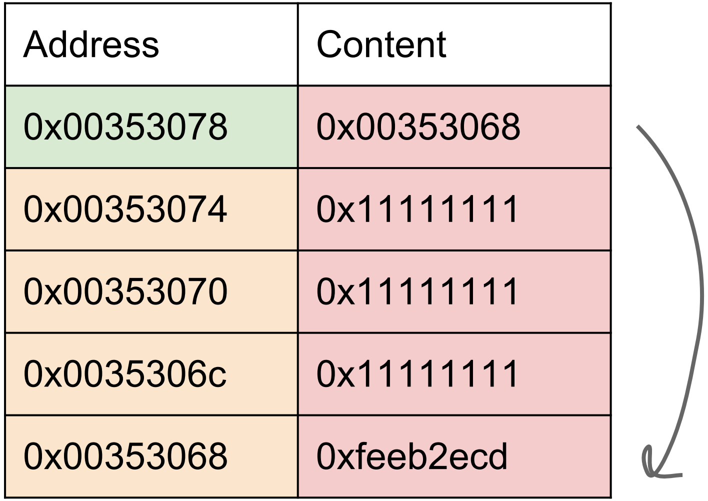
Here the buff is holding “file://foobar”

Corrupted!



	Address	Content
cmp	0x00353078	0x00353068
	0x00353074	0x11111111
	0x00353070	0x11111111
buff	0x0035306c	0x11111111
	0x00353068	0xfeeb2ecd

Here the buff is holding an attack payload



Address	Content
0x00353078	0x00353068
0x00353074	0x11111111
0x00353070	0x11111111
0x0035306c	0x11111111
0x00353068	0xfeeb2ecd

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Attack 2: Heap-based Buffer Overflows

- related heap objects are often allocated adjacently
- heap metadata can get corrupted

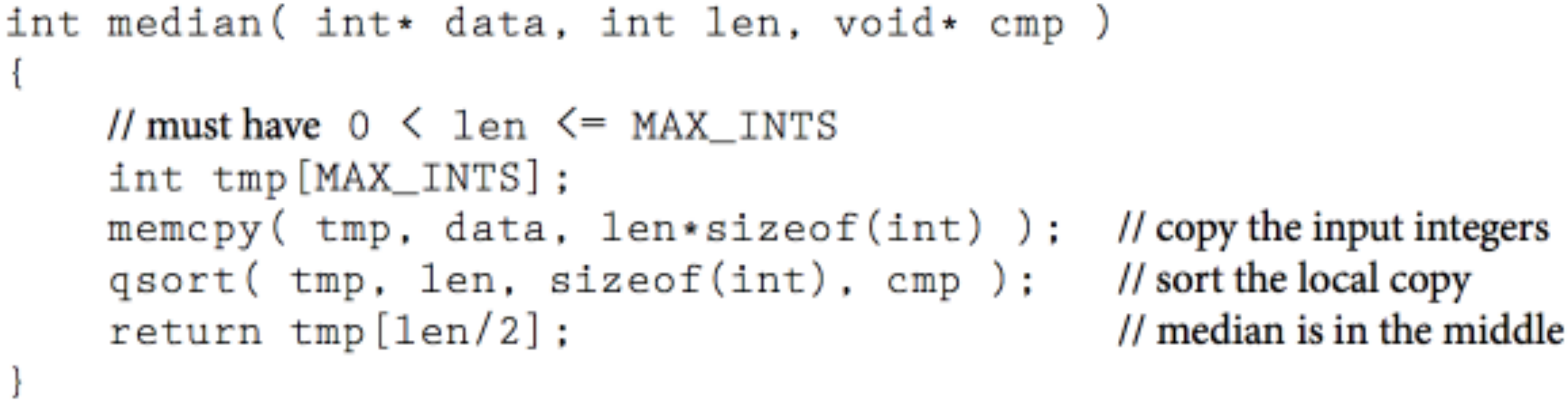
- Caveats:
  - trickier for attacker to determine heap addresses
  - relies on contiguous memory layout

- Direct Code Injection
  - input data contains attack payload and attacker directly manipulates instruction pointer to execute it
- Indirect Code Injection
  - input data contains attack payload but attacker uses existing software functions to execute it

Attack 3: Jump/Return-to-libc Attack

The attacker uses libc functions to execute desired machine code

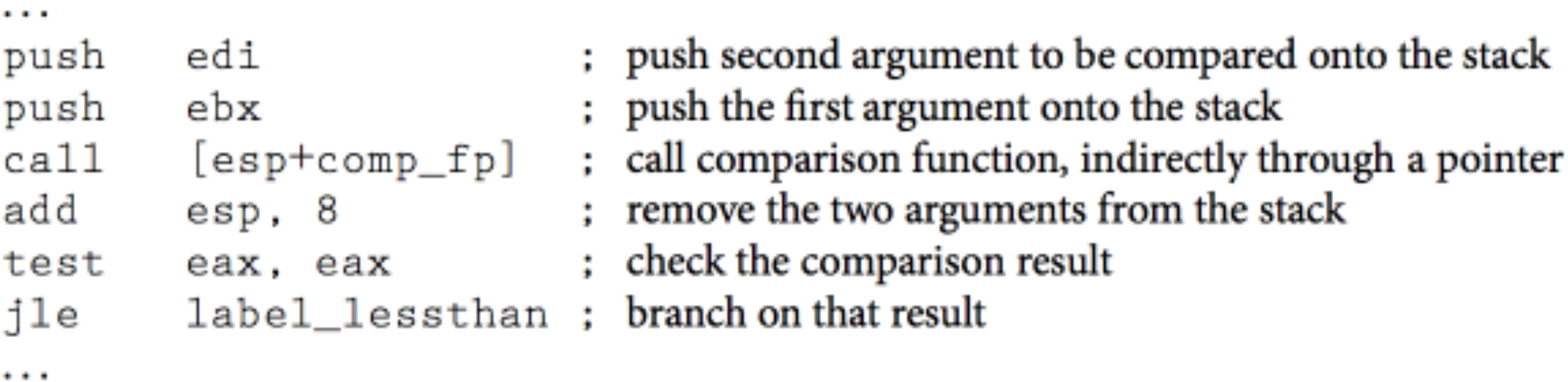
These useful bits of libc functions are called *trampolines*



```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

qsort is going to call cmp via a function pointer. What if we corrupt this function pointer?!

qsort(tmp, len, sizeof(int), cmp);

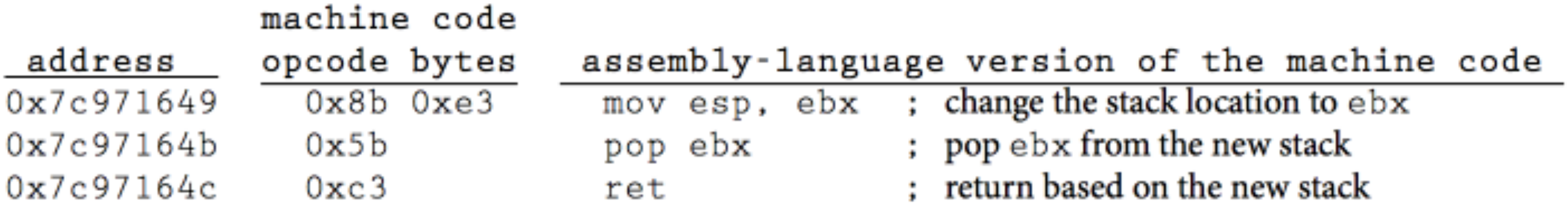


```
...
push    edi             ; push second argument to be compared onto the stack
push    ebx             ; push the first argument onto the stack
call    [esp+comp_fp]   ; call comparison function, indirectly through a pointer
add     esp, 8          ; remove the two arguments from the stack
test    eax, eax        ; check the comparison result
jle     label_lessthan  ; branch on that result
...
```

Notice that tmp is in %ebx

stack address	normal stack contents	benign overflow contents	malicious overflow contents		
0x0012ff38	0x004013e0	0x1111110d	0x7c971649	;	`cmp` argument
0x0012ff34	0x00000001	0x1111110c	0x1111110c	;	`len` argument
0x0012ff30	0x00353050	0x1111110b	0x1111110b	;	`data` argument
0x0012ff2c	0x00401528	0x1111110a	0xfeeb2ecd	;	return address
0x0012ff28	0x0012ff4c	0x11111109	0x70000000	;	saved base pointer
0x0012ff24	0x00000000	0x11111108	0x70000000	;	`tmp` final 4 bytes
0x0012ff20	0x00000000	0x11111107	0x00000040	;	`tmp` continues
0x0012ff1c	0x00000000	0x11111106	0x00003000	;	`tmp` continues
0x0012ff18	0x00000000	0x11111105	0x00001000	;	`tmp` continues
0x0012ff14	0x00000000	0x11111104	0x70000000	;	`tmp` continues
0x0012ff10	0x00000000	0x11111103	0x7c80978e	;	`tmp` continues
0x0012ff0c	0x00000000	0x11111102	0x7c809a51	;	`tmp` continues
0x0012ff08	0x00000000	0x11111101	0x11111101	;	`tmp` buffer starts
0x0012ff04	0x00000004	0x00000040	0x00000040	;	`memcpy` length argument
0x0012ff00	0x00353050	0x00353050	0x00353050	;	`memcpy` source argument
0x0012fefc	0x0012ff08	0x0012ff08	0x0012ff08	;	`memcpy` destination arg.

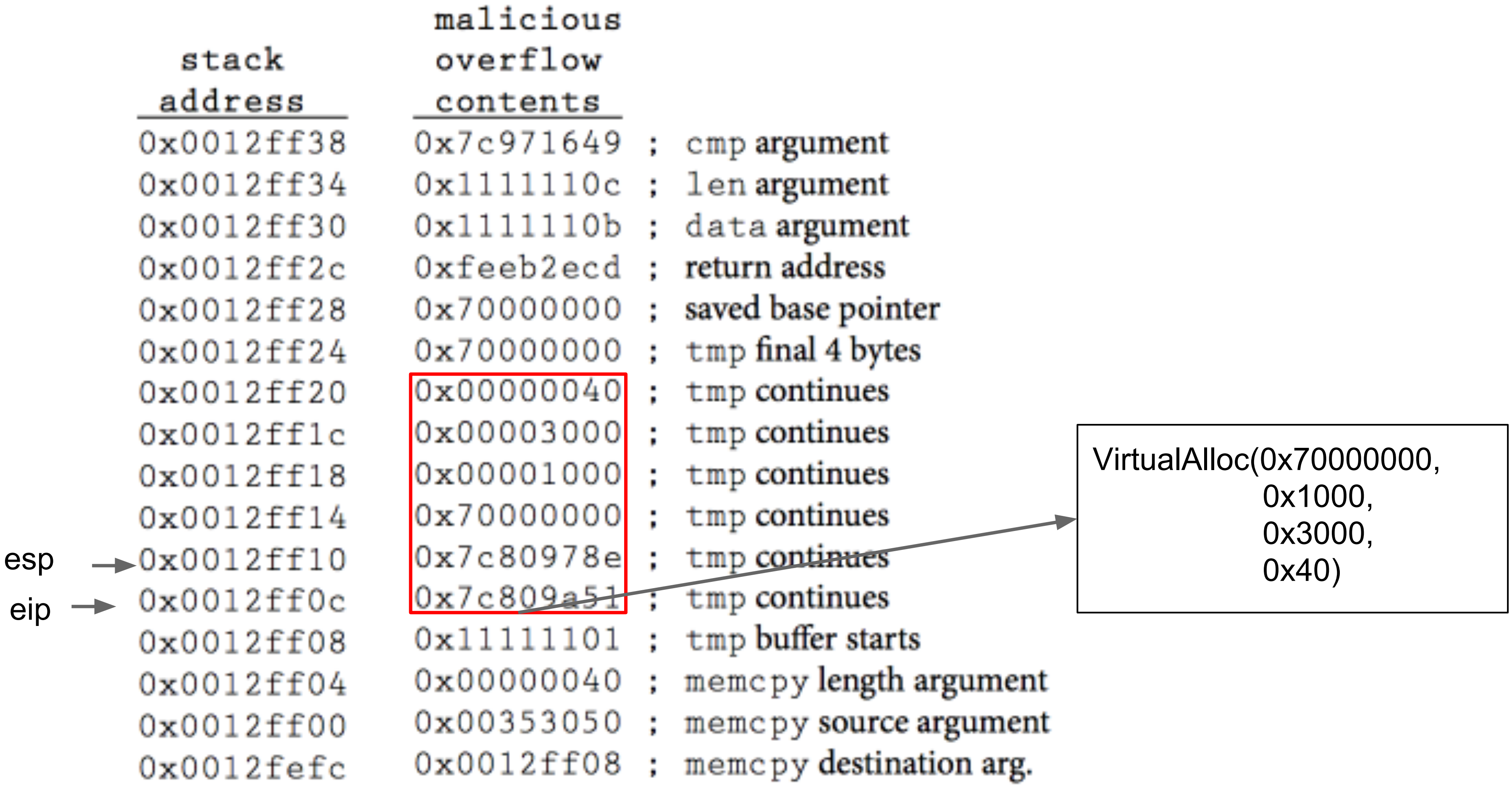
The corrupted cmp function points to a *trampoline*...



address	machine code opcode bytes	assembly-language version of the machine code
`0x7c971649`	`0x8b 0xe3`	`mov esp, ebx` ; change the stack location to `ebx`
`0x7c97164b`	`0x5b`	`pop ebx` ; pop `ebx` from the new stack
`0x7c97164c`	`0xc3`	`ret` ; return based on the new stack

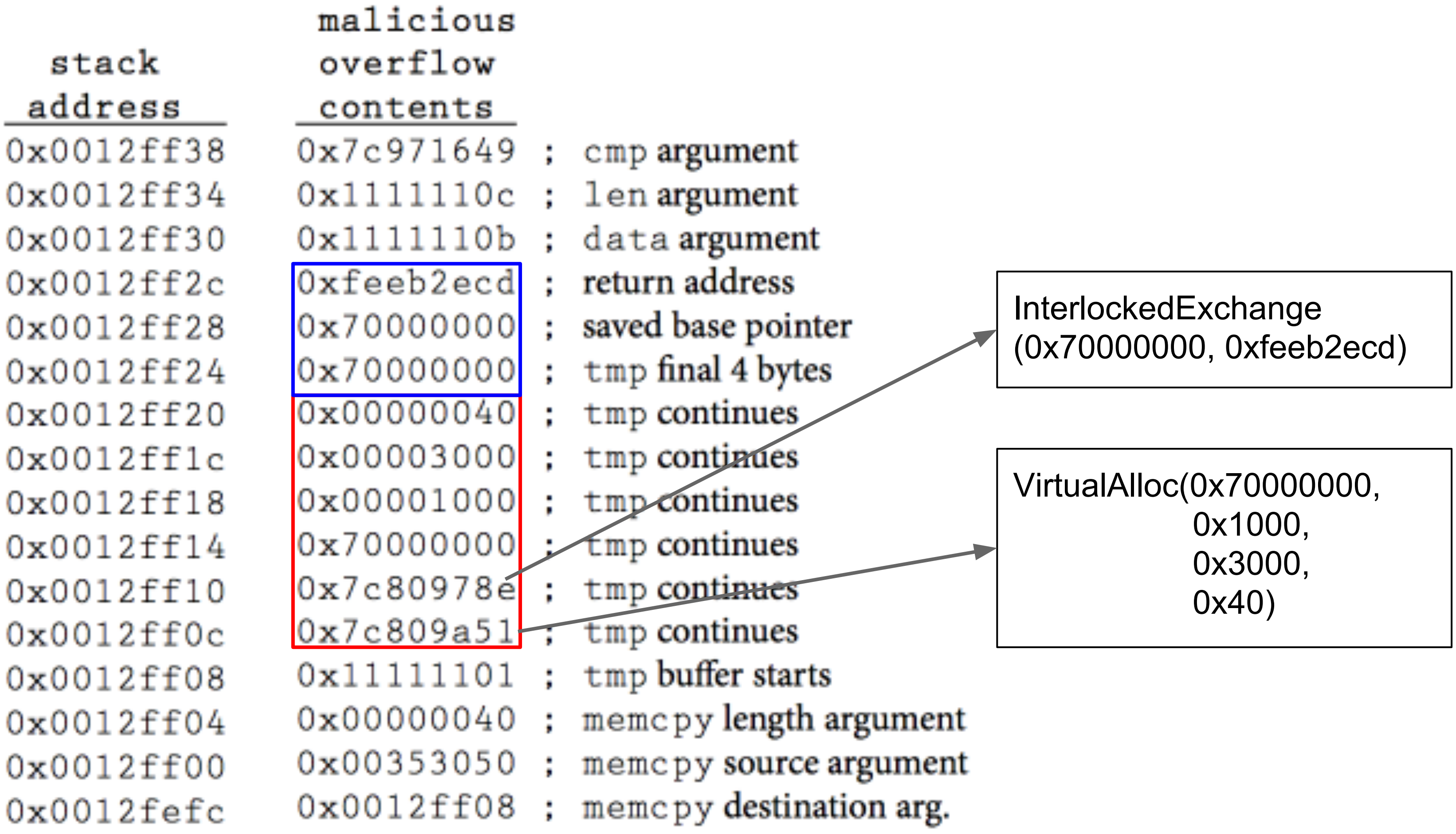
Remember tmp was in %ebx!
So this code:

1. sets stack pointer to the start of the tmp
2. reads a value from tmp
3. moves instruction pointer to second index of tmp



	stack address	malicious overflow contents	
	0x0012ff38	0x7c971649	; cmp argument
	0x0012ff34	0x1111110c	; len argument
	0x0012ff30	0x1111110b	; data argument
	0x0012ff2c	0xfeeb2ecd	; return address
	0x0012ff28	0x70000000	; saved base pointer
	0x0012ff24	0x70000000	; tmp final 4 bytes
	0x0012ff20	0x00000040	; tmp continues
	0x0012ff1c	0x00003000	; tmp continues
	0x0012ff18	0x00001000	; tmp continues
	0x0012ff14	0x70000000	; tmp continues
esp →	0x0012ff10	0x7c80978e	; tmp continues
eip →	0x0012ff0c	0x7c809a51	; tmp continues
	0x0012ff08	0x11111101	; tmp buffer starts
	0x0012ff04	0x00000040	; memcpy length argument
	0x0012ff00	0x00353050	; memcpy source argument
	0x0012fefc	0x0012ff08	; memcpy destination arg.

```
VirtualAlloc(0x70000000,
             0x1000,
             0x3000,
             0x40)
```



```
  stack        malicious
                overflow
 address        contents
0x0012ff38     0x7c971649  ;  cmp argument
0x0012ff34     0x1111110c  ;  len argument
0x0012ff30     0x1111110b  ;  data argument
0x0012ff2c     0xfeeb2ecd  ;  return address
0x0012ff28     0x70000000  ;  saved base pointer
0x0012ff24     0x70000000  ;  tmp final 4 bytes
0x0012ff20     0x00000040  ;  tmp continues
0x0012ff1c     0x00003000  ;  tmp continues
0x0012ff18     0x00001000  ;  tmp continues
0x0012ff14     0x70000000  ;  tmp continues
0x0012ff10     0x7c80978e  ;  tmp continues
0x0012ff0c     0x7c809a51  ;  tmp continues
0x0012ff08     0x11111101  ;  tmp buffer starts
0x0012ff04     0x00000040  ;  memcpy length argument
0x0012ff00     0x00353050  ;  memcpy source argument
0x0012fefc     0x0012ff08  ;  memcpy destination arg.
```

InterlockedExchange
(0x70000000, 0xfeeb2ecd)

VirtualAlloc(0x70000000,
0x1000,
0x3000,
0x40)

Attack 3: Jump-to-libc Attack

- Often targets the System func
- Often no new process launched -- Why is this a good thing?

Caveats:

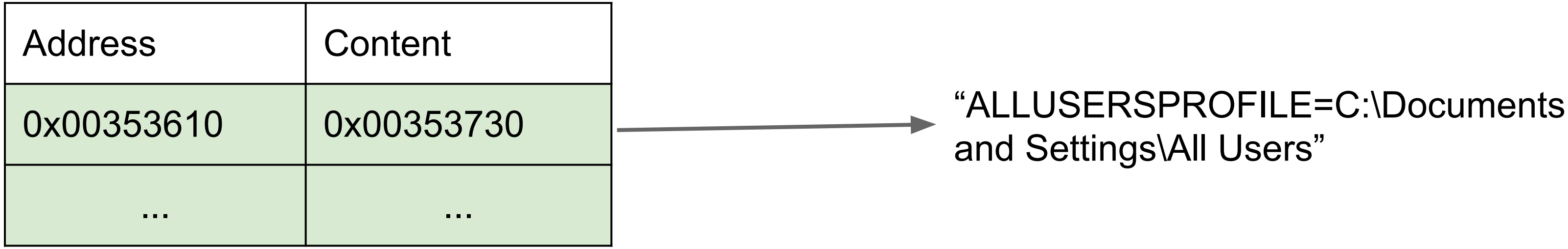
- Need access to library source code
    - even then versions and exec envs can vary

Attack 4: Data Corruption Attack

Modify data that controls behavior without using direct/indirect diversion from regular execution

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

Environment String Table



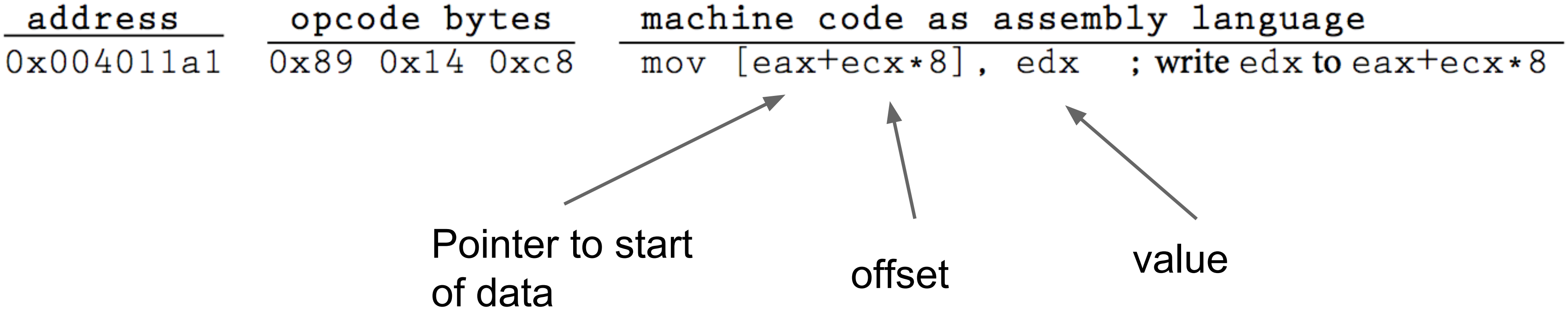
Address	Content
0x00353610	0x00353730
...	...

“ALLUSERSPROFILE=C:\Documents and Settings\All Users”

getenv() routine grabs a string from the environment string table to be passed to the *system*() routine.

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

data[offset].argument = value



address	opcode bytes	machine code as assembly language
0x004011a1	0x89 0x14 0xc8	mov [eax+ecx*8], edx ; write edx to eax+ecx*8

Pointer to start of data

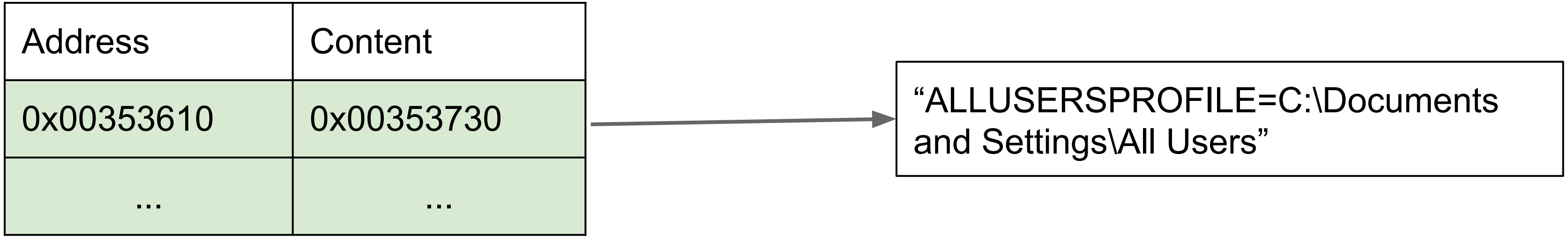
offset

value

If offset = 0x1ffea046 and if data = 0x004033e0
data addr + 8 * offset = 0x00353610 which is the first environment string pointer!

So we are essentially setting address 0x00353610 to our value=0x00354b20

Environment String Table



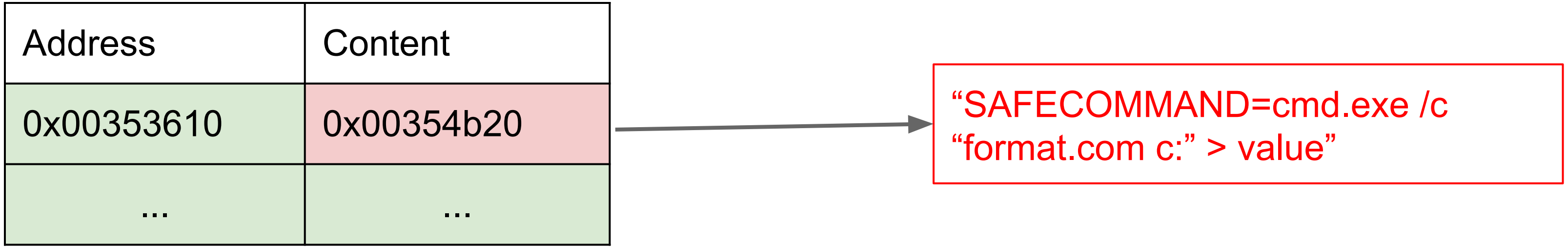
Address	Content
0x00353610	0x00353730
...	...

“ALLUSERSPROFILE=C:\Documents and Settings\All Users”

If we set
0x00353610 to our value=0x00354b20

getenv() routine grabs the string from the environment string table to be passed to the *system*() routine.

Environment String Table



Address	Content
0x00353610	0x00354b20
...	...

"SAFECOMMAND=cmd.exe /c "format.com c:" > value"

If we set
0x00353610 to our value=0x00354b20

getenv() routine grabs the string from the environment string table to be passed to the *system*() routine.

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

Attack 4: Data Corruption Attack

Caveats:

- Not all data is corruptible or fully corruptible
- Depends on how SW handles input
  - diff between corrupting input data for a calculator vs a command interpreter
- Not very useful by itself

Defense 1: Stack Canary

What’s the purpose of the canary?

Defense 1: Stack Canary

- Ideally....encrypt the return addresses!
  - but this is expensive

- Put a canary value above buffer on the stack
  - when function exits, check canary

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

Address	Content
0x0012ff5c	Arg two pointer
0x0012ff58	Arg one pointer
0x0012ff54	Return Address
0x0012ff50	Saved Base Pointer
0x0012ff4c	All zero canary value
0x0012ff48	Tmp Array (end)
0x0012ff44	
0x0012ff40	
0x0012ff3c	Tmp Array (start)

Defense 1: Stack Canary

- Why can’t the attacker just imitate the stack canary?

- Which of the 4 attacks will this defend against?

Defense 1: Stack Canary

- Why can’t the attacker just imitate the stack canary?
  - sometimes they can!
  - but often contains null bytes or newline characters
  - and/or uses a randomized cookie (harder to guess)
- Which of the 4 attacks will this work against?
  - Just stack-overflow, but can’t always defend
- Unfortunately has overhead

Defense 2: Non-executable Data

- Make data memory non-executable
    - this is now the norm!

- Which attacks might this prevent?

Defense 2: Non-executable Data

- Make data memory non-executable
    - this is now the norm!
- Which attacks might this prevent?
    - Attacks 1 & 2 fail
        - knows not to interpret machine op codes as instructions
    - Doesn’t defend against 3 & 4 -- why?

Defense 3: Control-Flow Integrity

- Expectations of higher-level software dictates rules for low-level hardware
  - ex. totally legal in low-level HW to jump to machine instruction in the middle of another op, but not the norm for higher-level SW
- When transfer control (i.e. via return statement or func pointer) check against restricted set of possibilities

Defense 3: Control-Flow Integrity

Caveats:

- Some overhead
- Can defend against attacks 1 & 2 & 3 but not 4

Defense 4: Address-Space Layout Randomization

Could also change layout in memory…

Why is this useful? What key assumption does this rely on?

Caveats:

- A bit of overhead
- Need a non-trivial shuffling algorithm!