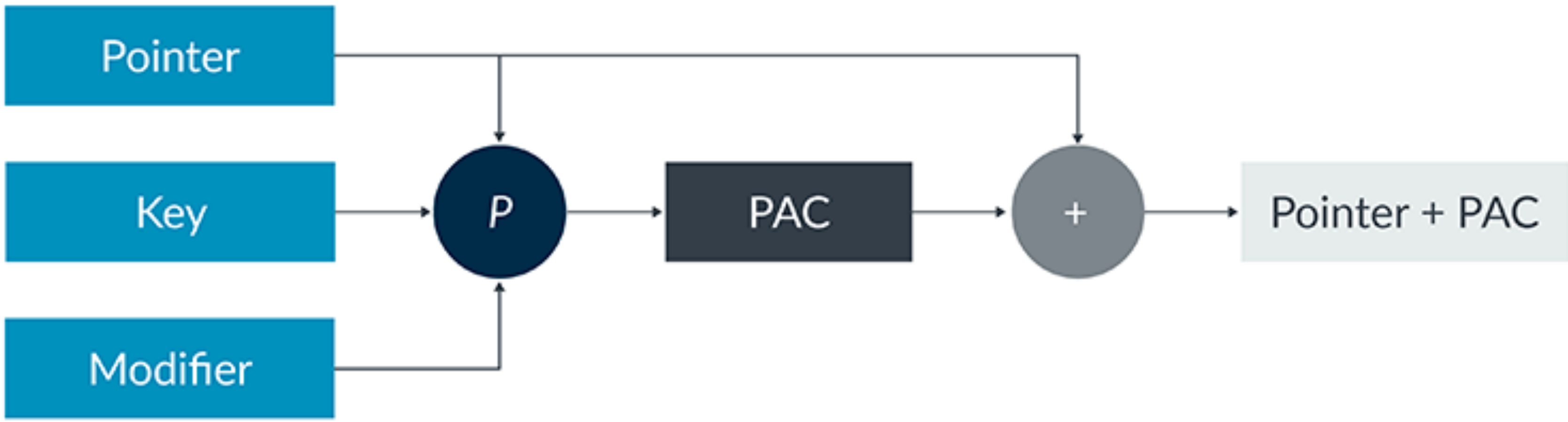
Backward-Edge Control Flow Integrity

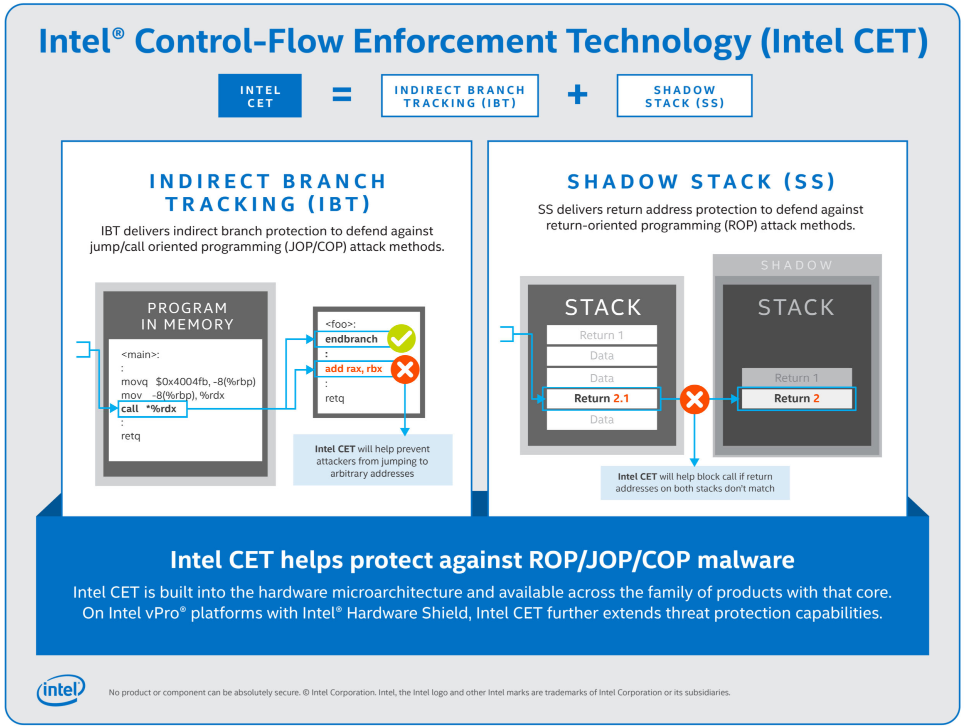
- Backward-Edge refers to callee-to-caller control flow
- E.g., Returns
- Many attacks that we learned in this course dealt with return address corruption
- i.e., Backward-edge control flow integrity VIOLATION through <u>memory corruption</u>

PA-based Return address signing on ARM



- $gcc -msign-return-address=all

Intel CET

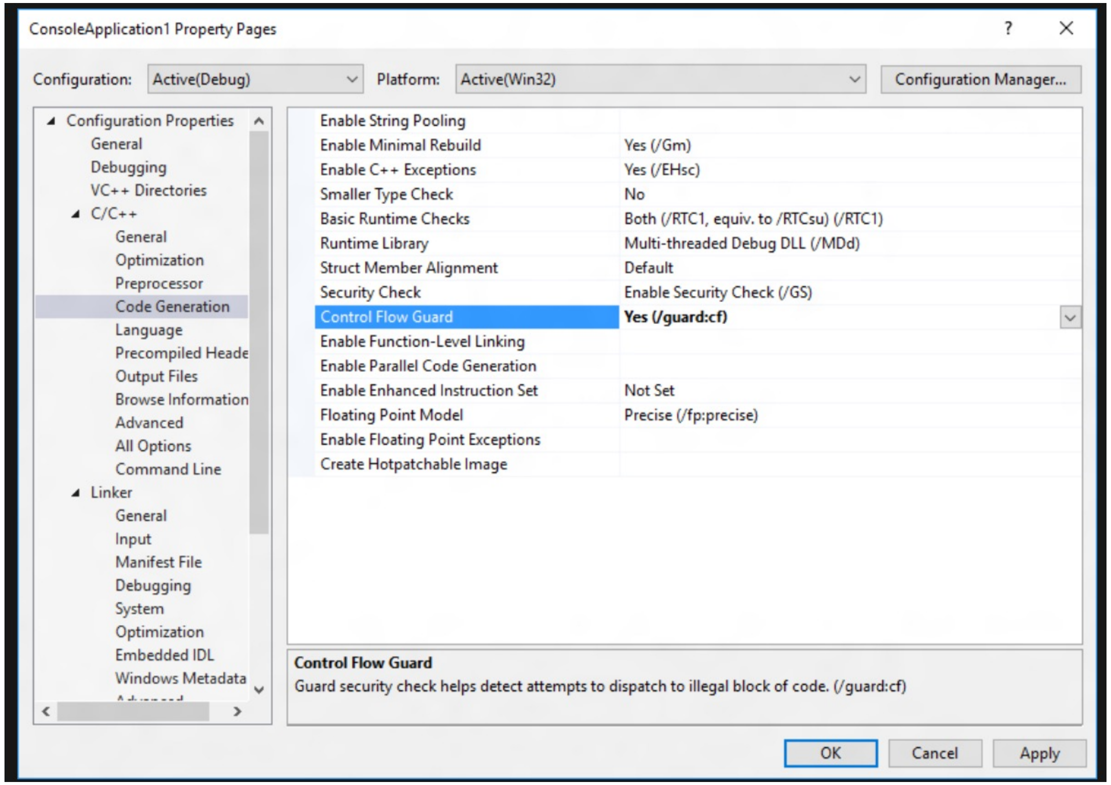


- ON by default (Since Intel 11th gen)

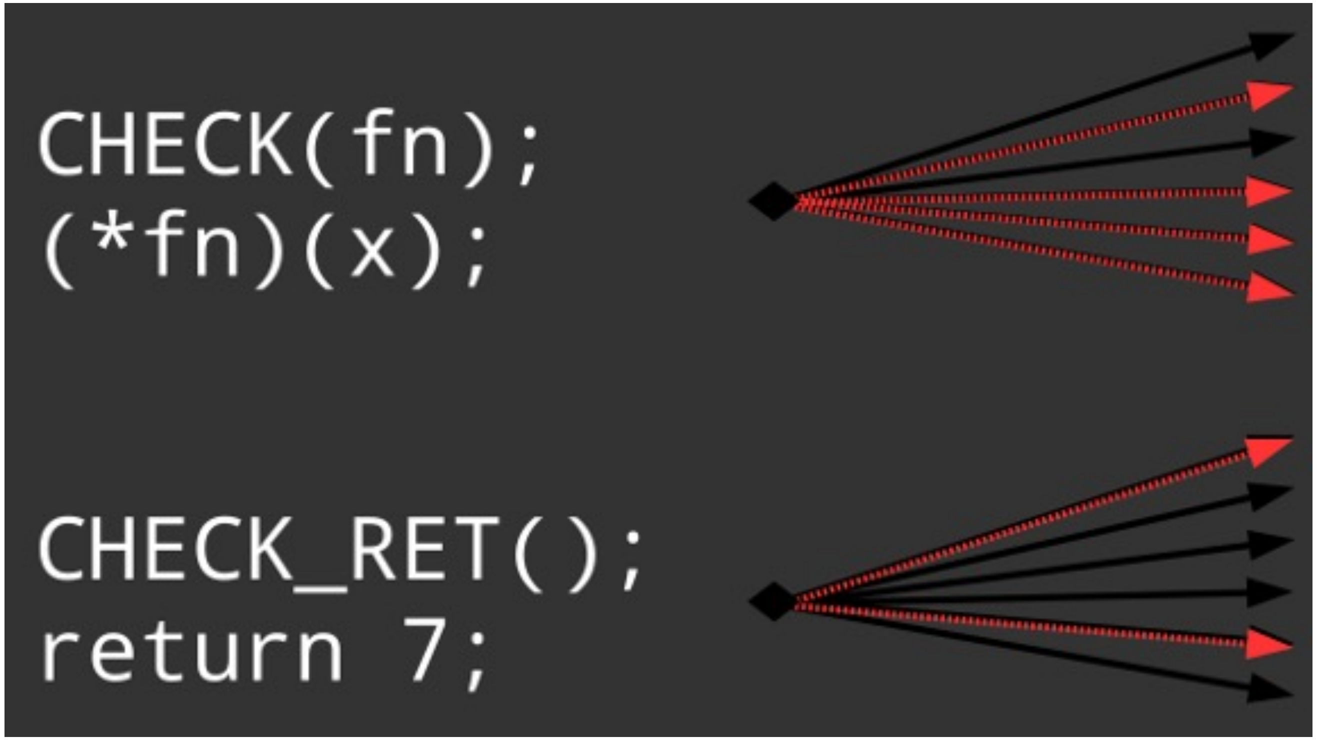
Forward-Edge Control Flow Integrity

- Forward-Edge refers to caller-to-callee control flow
- E.g., indirect function calls
- Function pointers can be corrupted via memory corruption
- Attackers can manipulate function pointers to call arbitrary functions

Microsoft Control Flow Guard



LLVM Forward-edge defenses (Type-based)



- clang –fsanitize=cfi-cfi-vcall
- clang –fsanitize=cfi-icall
- …

Secure Coding in C/C++

Dr. Kamil Sarac

2012 REU (Research Experiences for Undergraduates)

Department of Computer Science

University of Texas at Dallas

Outline

- Objective
- What is secure coding?
- Why should I care?
- Bug classes
    - Memory corruption
    - Integer overflows/underflows
    - Type conversion
    - Others
- Source code auditing
- Resources

Objective

- Be able to identify certain types of bugs/vulnerabilities in C/C++ source code
- Identify the correct way to avoid these bugs

Secure Coding

- Programming in a way as to avoid bugs and possible security vulnerabilities at the time of development, rather than reviewing and fixing code after the fact.

Why should I care?

- Create better software
  - Secure
  - Reliable
  - Extendable
  - …
- Save $$$
  - Cost more to patch and roll out than to prevent
  - Negative publicity can't be good...

Memory Corruption

- “...the contents of a memory location are unintentionally modified due to programming errors... When the corrupted memory contents are used later in the computer program, it leads either to program crash or to strange and bizarre program behavior.” - wikipedia

Common Culprits

- String functions that assign/copy values, without a length parameter
  - strcpy
  - strcat
  - sprintf
  - etc.
- These functions continue execution until a null byte is found in the source string
- Trusting the source of input to contain a properly terminated null string can be abused by attackers

Behind the scenes

```
char s[] = "hello"
```

Is equivalent to, and what the true representation of s looks like :

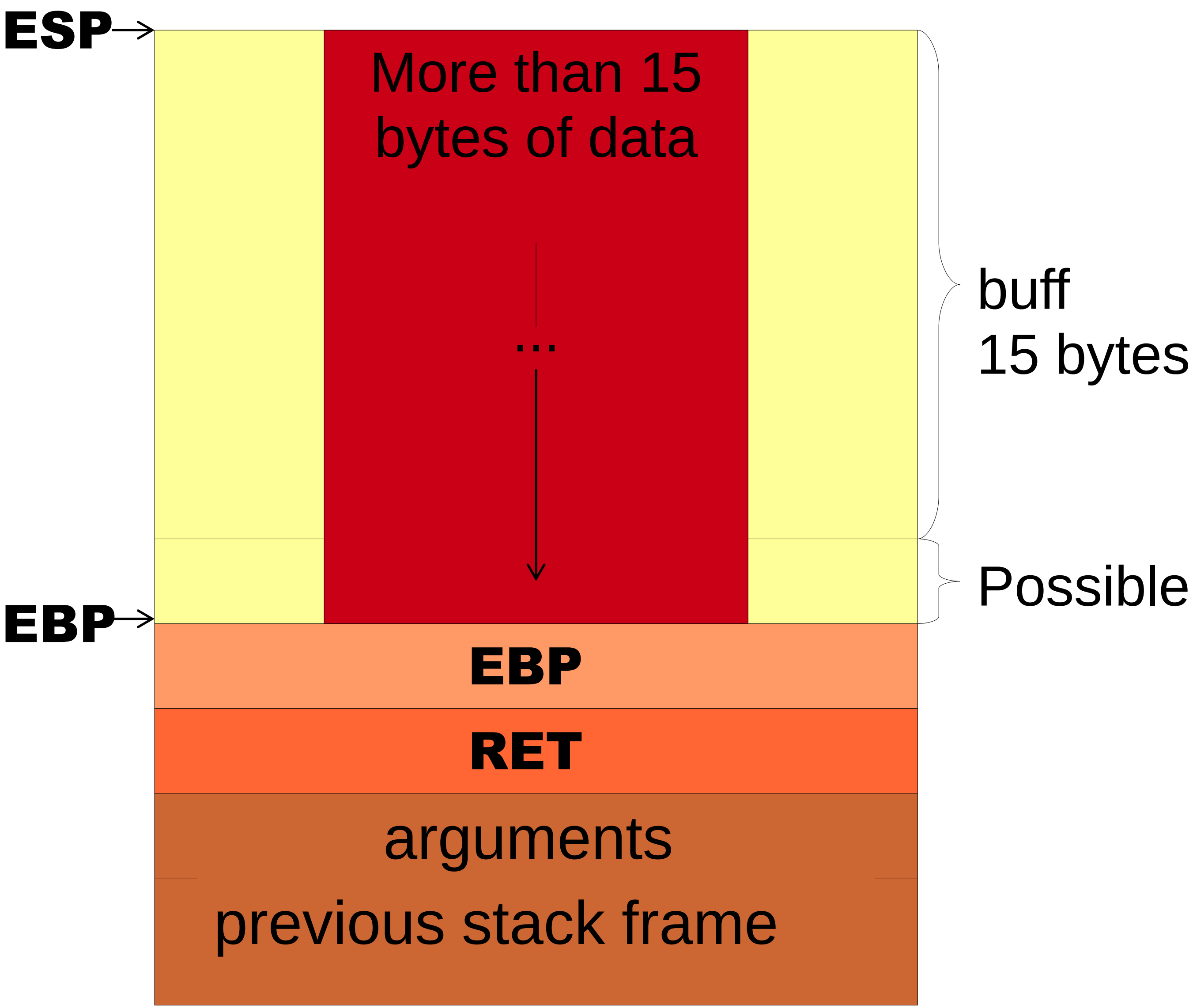
```
char s[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Behind the scenes...

```
int vuln_fuct(char *user_buffer) {
    char buff[15];
    strcpy(buff, user_buffer);
    ...
}
```

- If user_buffer is less than 15 bytes, there is no problem, but with 15 or more bytes strcpy will continue writing to the boundary of buff and down the stack – regardless of buff's size.

What's really going on



- strcpy will continue writing down the stack

Slightly Safer Functions

- These functions have a similar prototype which provides a length parameter :
  - strncpy
  - strncat
  - snprintf
  - …
- The 'n' indicates these functions take a length parameter.
  - strcpy(dest, source)
  - strncpy(dest, source, **length**)

I'm safe now... right?

```
int vuln_func(char *user_buffer){
    char buff[15];
    strncpy(buff, user_buffer, sizeof(user_buffer));
    ...
}
```

I'm safe now... right?

```c
int vuln_func(char *user_buffer){
    char buff[15];
    strncpy(buff, user_buffer, sizeof(user_buffer));
    ...
}
```

- Nope. strncpy is used and a specific length is being copied into buff this time, but it is the length of the untrusted user_buffer – not the destination buff.

I'm safe now... right? #2

```c
int vuln_func(char *user_buffer){
    char buff[15];
    strncpy(buff, user_buffer, sizeof(buff));
    ...
}
```

I'm safe now... right? #2

```
int vuln_func(char *user_buffer){
    char buff[15];
    strncpy(buff, user_buffer, sizeof(buff));
    ...
}
```

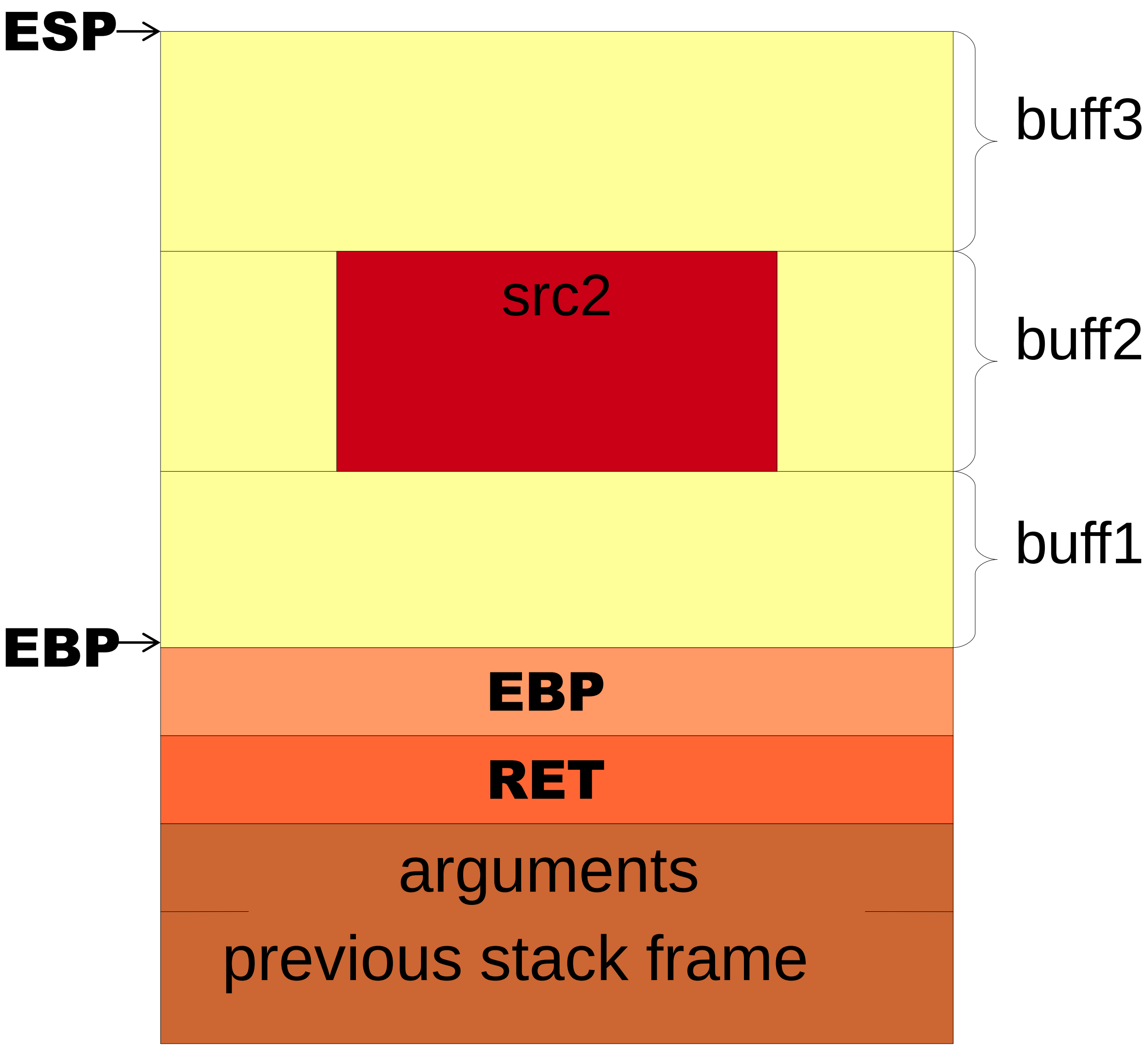
- Not quite, the buffer is not null terminated, the entire buffer could get copied into and filled up, not leaving room for a null byte
- What's the risk?

Forgetting null termination

```
int vuln_func(char *src2, char *src3) {
    char buff1[512];
    char buff2[512];
    char buff3[512];

    strncpy(buff2, src2, sizeof(buff2));
    strncpy(buff3, src3, sizeof(buff3));
    ...
    strcpy(buff1, buff3);
}
```

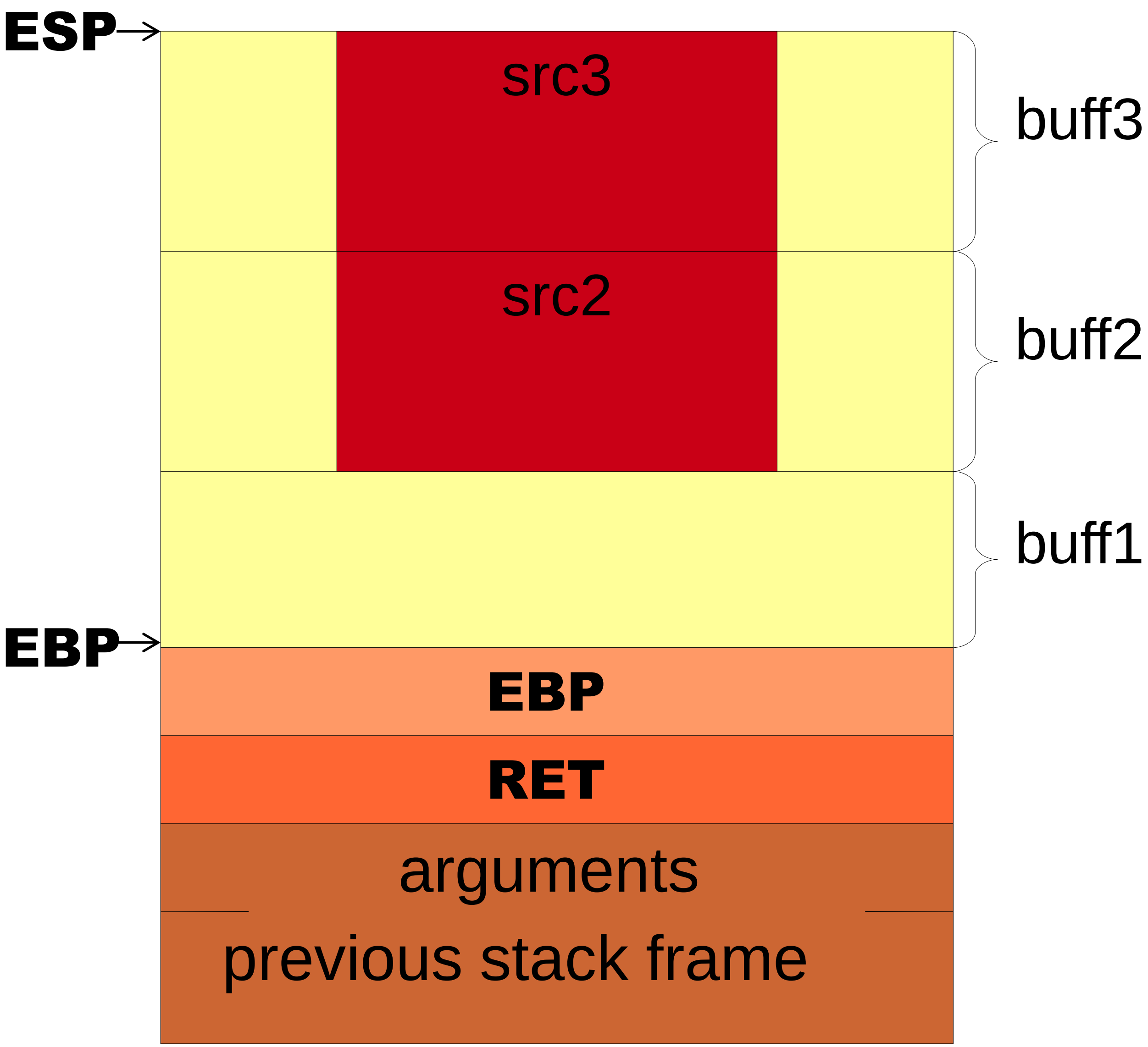
Forgetting null termination...




```
strncpy(buff2, src2, sizeof(buff2));
```

- If src2 is 512 bytes like buff2, it will not be null terminated

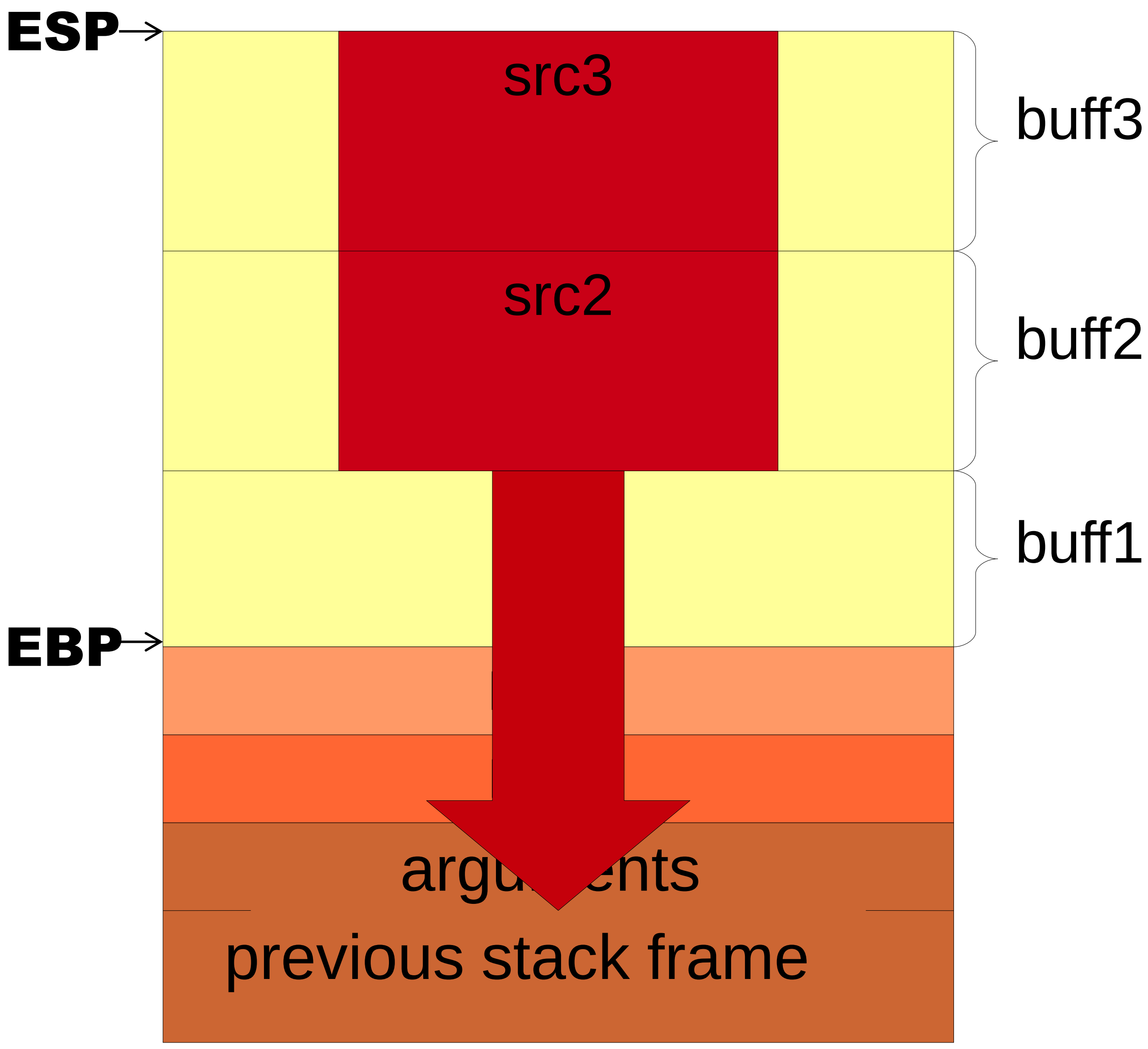
Forgetting null termination...




```
strncpy(buff3, src3, sizeof(buff3));
```

- If src3 is 512 bytes like buff3, it will not be null terminated
- The size of buff3 is now 1024 bytes

Forgetting null termination...




```
strcpy(buff1, buff3);
```

- Any string operation on buff3 will now operate on at least 1024 bytes, because no null bytes are in buff3 and buff2

The proper way

```
strncpy(dest, source, sizeof(dest)-1);
```

- Will not overrun dest
- Allows for null termination

The proper way... strncat

```
strncat(dest, source, sizeof(dest)-strlen(dest)-1);
```

- It's ugly, but correct
- If strncat isn't done properly, its another common culprit – check it

Integer Overflows/Underflows

- "...an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space" - wikipedia

Unsigned values

Type	Bits	Max Value	Max Value
char	8	2^8 - 1	255
short	16	2^16 - 1	65,535
int	32	2^32 - 1	4,294,967,295

Inter Overflows/Underflows...

- When a data-type is assigned a value larger than it's maximum size, the value will 'wrap' around

```
unsigned char c = 255;
c += 5;
```

c = $\boxed{1\,1\,1\,1\,1\,1\,1\,1}$ = 255

+5

c = $\boxed{0\,0\,0\,0\,0\,1\,0\,0}$ = 4

Overflow + allocation

```
int vuln_func2(int len, char *user_buff) {
    char *buff;

    buff = malloc(len+1);
    strncpy(buff, user_buff, len);
}
```

- The +1 could be for the null byte
- malloc takes an unsigned int as its length param
- If len is the max value of an unsigned int, 2^32-1, the length will wrap and malloc will allocate 0 bytes, but then 2^32-1 bytes will be written

Underflow

```
char buff[20];
char source1[] = "AAAAAAAAAAAAAAAAAAAA"; // 20
char source2[] = "BBBBBBBBBB"; // 10

strncpy(buff, source1, sizeof(buff));
strncat(buff, source2, sizeof(buff)-strlen(buff)-1);
```

- Buff will be completely filled, with no null termination from strncpy, which results in :

  strncat(buff, source2, 20-**20**-1)

  strncat(buff,source2, 2^32-1)

Rolls around to a very large number

Type Conversion

- “...implicitly or explicitly, changing an entity of one data type into another.” - wikipedia
- Arithmetic operations, assignments and comparisons cause type promotion and sometimes conversion

Type Conversion...

- Operations of data types (on x86) less than a signed int, will cause the promotion of the data types to signed ints – then the operation will take place – then the data types will be demoted to their original values
- Unless an unsigned int is in the operation, then the other value will promoted to an unsigned int
- Why? On x86 systems integers are assumed to be the most efficient data type

Why is this a problem?

```c
#define MAXLENGTH 1024
...

int vuln_network_func(int sock) {
    char buff[MAXLENGTH];
    int size;

    // first 2 bytes determine size
    size = read(sock, buff, 2);

    // sanity check
    if (size > MAXLENGTH)
        // error
    else
        read(sock, buff, size);
    ...
}
```

- MAXLENGTH is a signed value, so is size.
- A size of -1 will pass the signed sanity check, then size is converted to a very large unsigned value in read

Things to look for

- Signed values being used as lengths
- Unsigned values being checked less than 0
  - Like return values, these checks will always get bypassed

Others

- String vulnerabilities make up a large portion of C/C++ bugs, but there are several others
  - Format strings
  - Off-by-one
  - etc.
- For the sake of sanity and time, we won't cover these

Actually...

- Reviewing other people's/project's source code with the intent of discovering vulnerabilities is it's own line of work...
- Source code auditing

What is source code auditing

- “...a comprehensive analysis of source code in a programming project with the intent of discovering bugs, security breaches or violations of programming conventions.” - wikipedia
- Analyzing source code in order to discover flaws

Source Code Auditing – who/why?

- Security researchers
  - Fame, fun, hobby
- Code auditors
  - career
- Exploit development - good and bad guys
  - High profile vulnerabilities and their exploits sell for high dollar $$$

Before you begin

- Scope
  - Pick your targets
    - Sources of input
    - Any form of parsing
    - Binary protocols (files, network, …)
  - Balance time and code
- Gain an understanding of the target
  - Documentation, manuals, etc.
- Don't forget the easy stuff
  - Comments! “fixme”, “bad”, etc.

Resources

- Open source projects are a great way to practice and hunt for vulnerabilities
- More specific / advanced tutorials are online
- https://www.fortify.com/vulncat/en/vulncat/index.html
  - Listing of vulnerability classes and types, by language