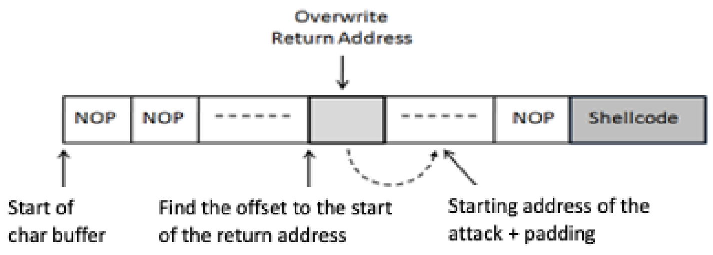
CS 88 Week 3: Class 6: Buffer Overflow Defenses

Discussion Question 1: Stack Canaries.
We saw that buffer overflow attacks work by overflowing the saved return address on the stack (Figure 1).


Overwrite Return Address
NOP
NOP

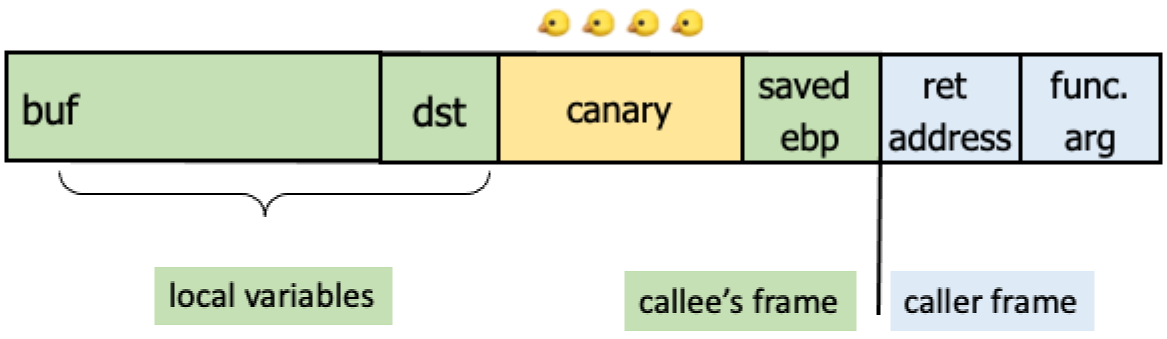
NOP
Shellcode
Start of char buffer
Find the offset to the start of the return address
Starting address of the attack + padding

Figure 1: showing a stack buffer overflow with no protections in place to defend against such an attack.

Subverting Stack Canaries: To prevent such attacks we can implement a stack canary as we saw in the pre-class video. Assume that we have the following stack layout shown below, with local variables, followed by the stack canary followed by the saved ebp and return address. Suppose our vulnerable code has the following declarations, and our local variables are stored on the stack as shown in the figure below.

```
char buf [500];
char * dst;
*dst = buf[0];
```

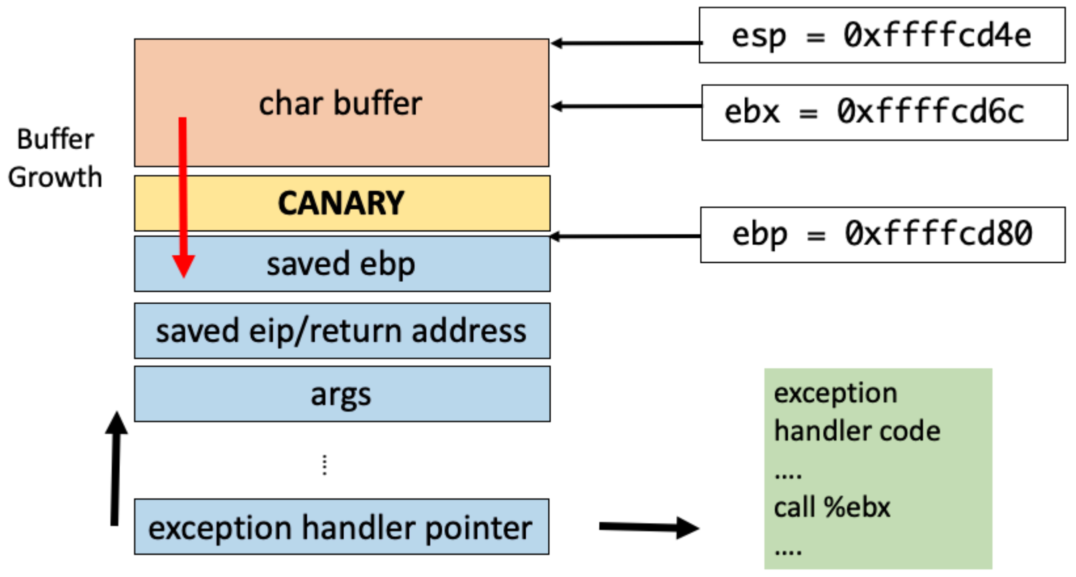
Assume that the attacker can write into the memory locations of both dst and buf. Design a mechanism that could still result in a shellcode exploit **without affecting the stack canary**.



Stack Canaries Part 2: Deploying the Code Red Worm.

The Code Red Worm was a computer worm that infected 400,000 machines in 6 days and was a 0-click remote exploit that leveraged buffer overflow vulnerabilities in Microsoft's IIS services (a.k.a. Web services). After the initial infection and incubation periods, Code Red was programmed to unleash a denial-of-service attack on the Whitehouse.gov Web site by targeting the actual Whitehouse.gov IP address.

Your task is to figure out how this worm leveraged buffer overflow vulnerabilities with stack canaries in place to execute a malicious payload. The information below sets up the attack. You can use the following figure as reference.



- A malicious HTTP request asks the webserver for an unusual URL containing the following payload:
  `GET /default.ida?NNNN[...]NNN%u9090%u6858`**`%ucbd3%u7801`**`%u9090%[...]`

- This HTTP request is stored by the webserver in a local char buffer on the stack using strcpy().

- The payload happens to be much bigger than the buffer and ends up overflowing the stack canary.

- The payload contains:
    - Series of Ns.
    - Malicious code
    - Series of Ns.
    - the characters %ucbd3%u7801 corresponding to a memory address that contains the instruction CALL EBX.

- Before the function returns, the stack canary overflow is detected by the compiler, which calls the exception handler pointer. The exception handler pointer is a function pointer that holds the **memory address of the starting instruction** of the exception handling code to be executed if stack smashing is detected.

In constructing your attack you can make the following assumptions:

- The payload can be of arbitrary length, and can therefore modify *all the contents of memory below the stack canary, including the exception handler pointer.*

➔ Assume that the ebx register when the overflow occurs, points to a known memory address within your buffer payload.

Draw out the steps to launch the attack in the figure above and describe the following items in your attack:

- Which key memory location on the stack would you need to place your malicious code to make this attack successful?

- Which memory location on the stack would you need to overflow to change the control flow such that your malicious code is executed instead of the exception handling code?

Now that you have seen two different types of attacks on stack canaries discuss the following in your group:

1) One class of attacks which the stack canary **will** prevent
2) Two general classes of attacks which the stack canary **will not** prevent

References:

1. https://nakedsecurity.sophos.com/2021/07/15/the-code-red-worm-20-years-on-what-have-we-learned/
2. https://en.wikipedia.org/wiki/Code_Red_(computer_worm)
3. https://cacm.acm.org/magazines/2001/12/7187-the-code-red-worm/fulltext

Discussion Question 2: Address Space Layout Randomization

ASLR aims to introduce randomization in memory addresses that make it hard to guess the layout of stack, heap and code regions on memory. We've seen that on 32 bit machines, we can randomize up to 24 bits of memory. However realistically, randomly mapped objects in memory can cause significant memory fragmentation. On average, this results in a 16 bit entropy on 32 bit systems, and a 40 bit entropy (randomization) on 64 bit machines.

Question 1: How much more entropy do we get on 64 bit machines compared to 32 bit machines (expressed in powers of 2). Do you think it is sufficient to prevent a brute-force attack?

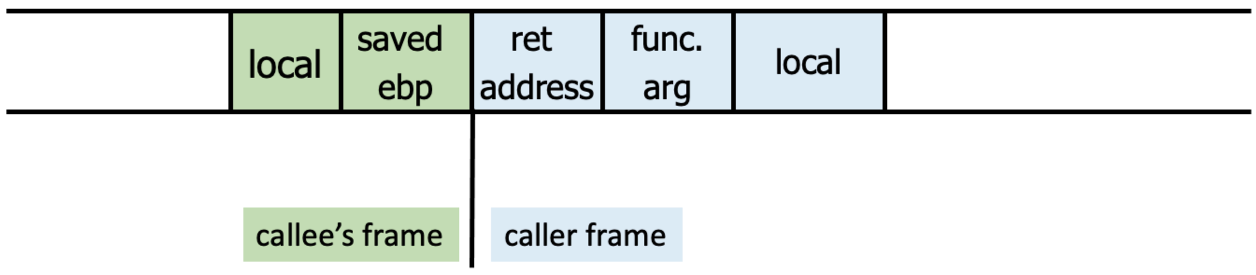
Question 2: If we combined ASLR with stack canaries which of our security principles are we applying:

(Circle all that apply)

A. Principle of Least Privilege:
B. Defense in Depth
C. Use Fail-Safe Defaults
D. Complete Mediation: check access to every object (No race conditions: Time of Check vs. Time of Use)

Discussion Question 3: Non-Executable Stack

A third line of defense we discussed in the pre-class videos was enlisting hardware support to mark all writeable memory locations as non-executable.



If this was the only defense we had in place, describe a method in which you could execute shellcode, or any arbitrary sequence of instructions, without being able to inject a single line of code into your stack or heap!

Hint: we can still get the return address to point to valid code instructions. Also the x86 instructions are arbitrarily long and don't have to start or end at 4 byte boundaries. As an example, consider the same instruction interpreted as an 8 byte and a 4 byte instruction - they end up forming completely different instructions!!

```
C7 45 d4 01 00 00 00 f7 : movl $0x00000001, -44(%ebp)
            00 00 00 f7 : add %dh, %bh
```

Discussion Question 4: Detecting Software Vulnerabilities & Reasoning about memory safety.

As software programmers, you can see how while we have multiple lines of defense, ultimately it just takes one badly written function to create a vulnerability that an attacker can exploit. In this discussion question we will discuss approaches that build ***confidence*** that our code executes in a memory safe and correct manner. The following is a form of **formal verification**.

Given this, Whose responsibility is it to write and check that code is memory safe?

A. The software developer using existing code bases
B. The software developer responsible for writing support libraries
C. The user of the software
D. Someone else (discuss)

Given the options you picked, how should the people you selected ensure that the code is safe to run? Should they understand the entire code base?

Formal Verification is a form of proving that your software does what you say it does, under certain operating assumptions. We will cover this at a high-level for memory safety. Our approach is going to be to build confidence function-by-function, and module-by-module. For each instruction, module and function in our code we should establish pre-and post-conditions. These terms are defined below:

- Pre-conditions: what must hold for a function/module/instruction to operate correctly
- Post-conditions: What holds after a function/module/instruction completes.

This also applies to individual code statements in the following manner:

- Code statement 1's postcondition should logically imply code statement 2's precondition.
- **Establishing Invariants**: We should check that invariants that are critical to the correctness properties of our code, always remain true at a given point in the function (important for loops!).

For each of these pre/post conditions and invariants, we want to express them in a manner that the person writing/reading/checking the code knows how to evaluate them.

Here's an example of a simply malloc function:

```
/* requires:→ pre-condition statement here */
/* ensures: → post-condition statement here */
void *mymalloc(int n){
      void *p = malloc(n);
      return p;
}
```

What would we want our pre/post conditions to be?

- Precondition: requires n to be an unsigned integer that cannot have integer overflows.
- Post-condition: requires p!= NULL and p is a valid pointer

```
void *mymalloc(unsigned int n){
      void *p = malloc(n);
      if (!p){
            perror("malloc");
            exit(1);
      }
      return p;
}
```

Task 1: Come up with the pre and post conditions for the following code, to ensure that it safely accesses memory:

```
int sum(int a[], int n) {
  int total = 0;
  for (int i=0; i<=n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition and post-conditions required
(3) Propagate requirement from the beginning of function to the return value.

Your solution:

```
/* requires:
   requires:
   ensures:
*/
int sum(int a[], unsigned int n) {
int total = 0;
 for (int i=0; i<n; i++){
    total += a[i];}
 return total;
}
```

Task 2: Come up with the pre and post conditions for the following code, to ensure that it safely accesses memory:

Here is the code commented:

```
char *tbl[N];     // array of char pointers i.e., array of string pointers


int hash(char *s) {            //char pointer
  int h = 17;
  while (*s)            //dereference pointer, if *s = '\0' = null value
                              //then, exit the loop.
      h = 257*h + (*s++) + 3; //multiply h with a large value, add *s, add 3,
                        //then increment s
  return h % N;               //return the value of a modulo operation.
}




int search(char *s) {
  int i = hash(s);                        // call the hash function
  return tbl[i] && (strcmp(tbl[i], s)==0);// compare two strings tbl[i] and s.
}

bool search(char *s) {
  int i = hash(s);                        // call the hash function
  return tbl[i] && (strcmp(tbl[i], s)==0);// compare two strings tbl[i] and s.
}
```

Q: If we ascertain that the code snippets are memory safe (or are modified to be memory safe), are we done?

A. No this still can have undefined behavior
B. Yes we are done!
C. Something else