These notes were written by students in CSE 127 in Winter 2021. They have been lightly edited but may still contain errors.

Buffer Overflow Defenses

From the last lecture, we know that buffer overflow attacks are based on overflowing an array or buffer with content that extends beyond its capacity. In this lecture we will be looking at different mitigations to the buffer overflow problem.

1 Avoid Unsafe Functions

One very obvious way is to avoid unsafe functions like `strcpy`, `strcat`, `gets`, `...`". We should try to not use those unsafe functions, but it requires extra work to rewrite the code manually, and there could be buffer overflow vulnerabilities in user-written functions. Besides, it is not possible to find substitutes for every unsafe function.

Buffer overflow attacks can happen in even the most basic functions. Consider the printf example:
`printf(''%s\n'', buf)` vs `printf(buf)`

While the former one might have vulnerabilities, it is the expected way of doing printf. However, many people get lazy and use the latter one. There are vulnerabilities with not formatting the printf. For example, what if the content of buf is a format string? If we let the content of `buf` be `%s%s%s`, the program will print what is allocated in the stack, usually args, and the attacker will be able to read sensitive data.

2 Stack Canaries

We cannot avoid buggy C code, and thus we need tactics to mitigate those errors. We just discussed how we should avoid unsafe functions, and now we will talk about stack canaries.

The word "canary" comes from mining in the old days. Miners will bring birds with them to detect toxic gases such as carbon monoxide and methane. Those toxic gases will affect the bird before affecting humans.

The idea behind a stack canary is to prevent control flow hijacking by detecting overflows. We put a canary between local variables and saved frame pointer and return address to prevent buffer overflowing into frame pointers and return address.

Consider the high-level example:

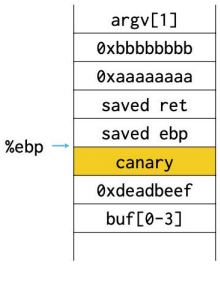
High-level example

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

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

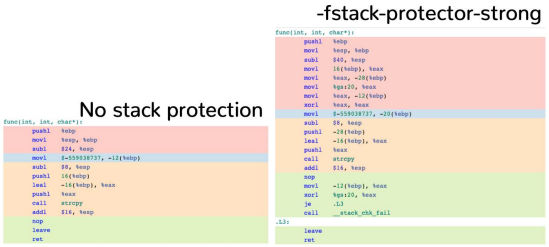


As we can see from the code, we use a stack canary to indicate the end of stack variables. If the attacker tries to perform a buffer overflow attack, the canary will be overwritten, and the program will detect the attack. When compiling, we can call "-fstack-protect-strong" to tell the compiler that we want a stack canary to protect the stack frames.

Take a look at the assembly code and notice the difference between a protected stack's assembly code and a non-protected stack's code.

Tradeoffs of stack canaries

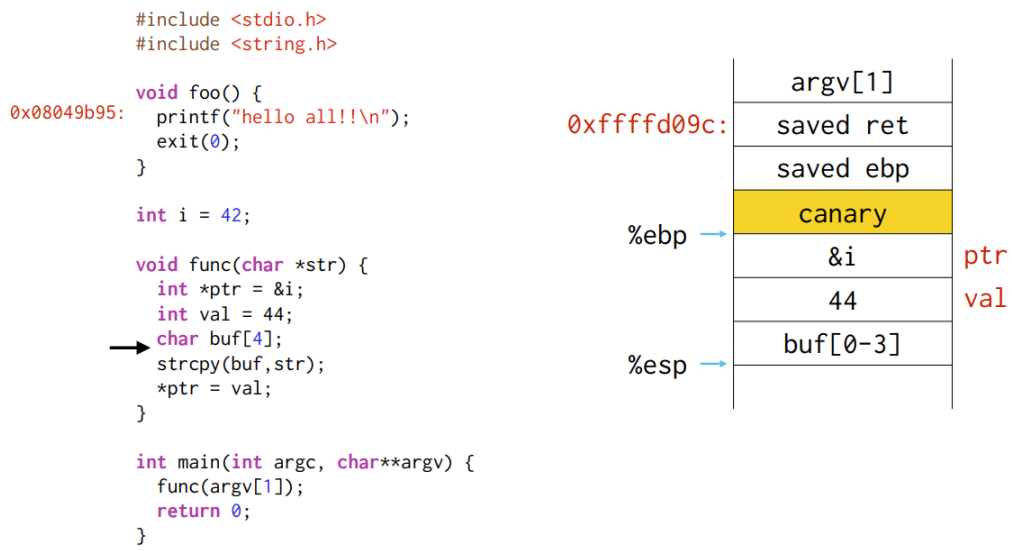
- **Easy to deploy:** Can implement mitigation as compiler pass (i.e. don't need to change your code)
- **Performance:** Every protected function is more expensive



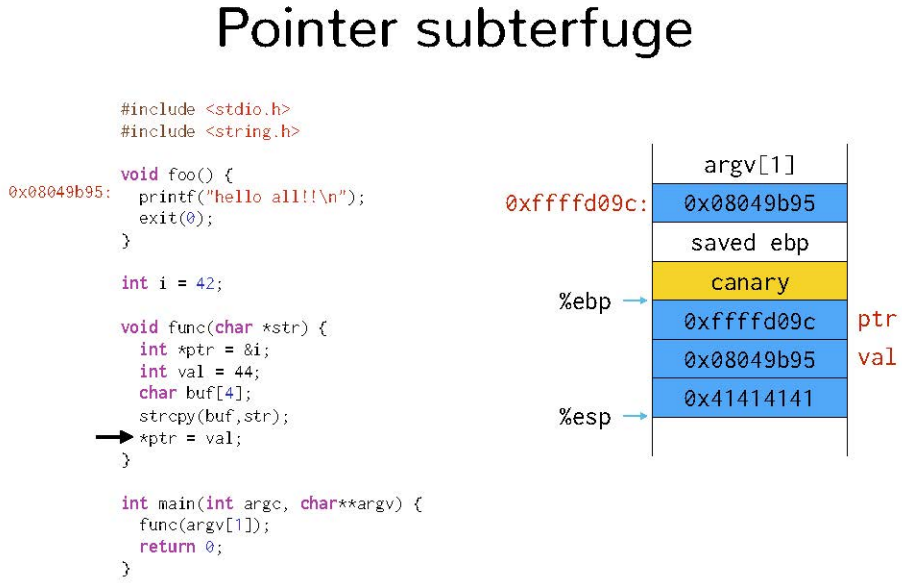
The trade-off with a stack canary is that although it protects stacks from overflow attacks most of the time, every protected stack is more expensive in terms of code length and complexity. It is also possible to go around stack canaries. There are four ways to get around with stack canary:

2.1 Pointer Subterfuge

Pointer subterfuge is a way to change the value of a local pointer on the stack to the malicious function's address. With pointer subterfuge, we can modify the return address of a function without overwriting the stack canary. Consider this example:



We should clearly see that strcpy is not a safe function. In this case, if the passed in str is too long, it will overflow buf and go into val and ptr. Take a look at the next picture specifying the address of the malicious function and return address of current stack:



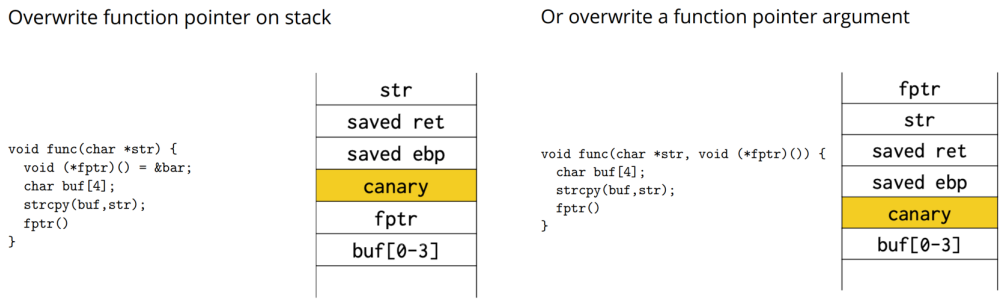
If we design the str's content in the way that it first overflows val with the address of the malicious function, and then it overflows ptr's content with the return address of current stack, when we do "*ptr = val" after the strcpy call, it will write the address of the malicious function into the return address of the current function. Thus, when the function ends and tries to return, it will call the malicious function. This method changes the return address without overflowing it, which means that it will not overflow the stack canary.

2.2 Overwriting the function pointer elsewhere on the stack

Function pointers on the stack are not guaranteed to be protected by a stack canary. An attacker may potentially override the buffer into the function pointer with a different value. Alternatively, the function pointer (fptr)

arguments could also be overridden.

Ultimately, when the function is invoked, the program will jump to the location determined by the new value of the function pointer, in which the attacker changed. The attacker can make a malicious section of code execute in this way since they can already manipulate the function pointer.



2.3 Buffer Overflow with Fixed Canary

Buffer overflows can occur due to various string and memory operations. We would need to pick a fixed canary value, which can be done in a few ways:

- Pick a random canary value (i.e. during runtime).
- Pick a clever canary value.
  - For example, 0x000d0aff (0, CR, NL, -1) to terminate string operations such as strcpy and gets
  - Even if the attacker knows the value, they cannot overwrite past canary!

2.4 Learning the Canary

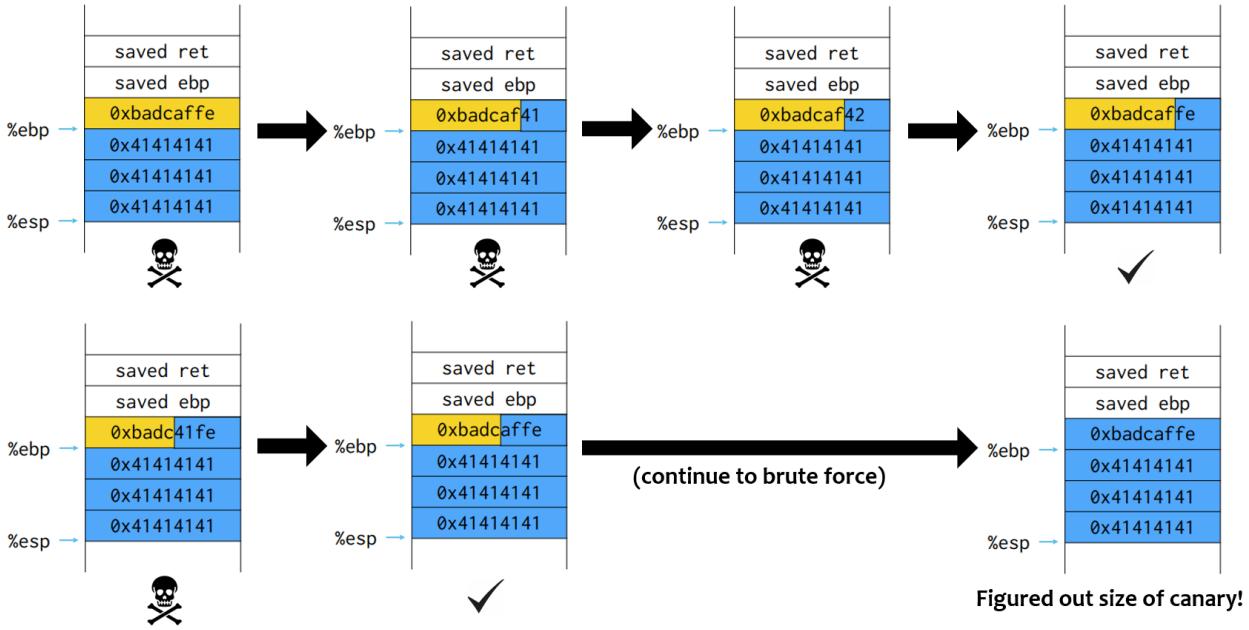
Here are two approaches to learning the stack canary:

Approach 1 - Chained Vulnerabilities: In this approach, we exploit a single vulnerability to read the canary value, and exploit a second vulnerability to perform a stack buffer overflow.

Approach 2 - Brute-Force Approach on Servers: Take a web server such as Apache2 or Nginx for example. We would have two types of processes: a main process and a worker process.

- For the main server process, we would establish a listening socket and fork several working processes off the current process.
- Each worker process would accept connections on the corresponding listening socket and process the web server request. Note that each worker process contains the same memory contents and layout as the parent process. This would include canary values.
- If any of the worker processes gets killed off, then a new worker process should be forked from the main server process. Fork on crash would let the end user test out different canary values.

Below is a demonstration of how an attacker attempts to brute-force a canary byte by byte. In essence, the attacker is overflowing the memory:

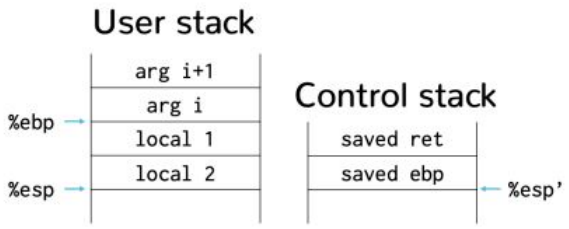


The attacker would know that the value of the canary is successful when the program does not crash when writing byte-by-byte to the stack canary. If the program crashes when writing an extra byte, then the guessed value of the canary is incorrect.

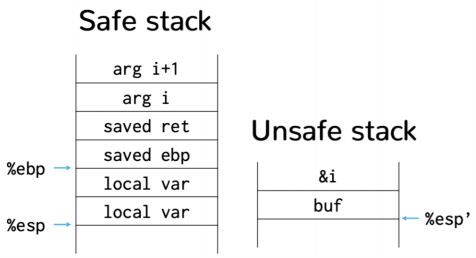
3 Separate Control Stack

A defense to these buffer overflow attacks would be to have Separate Control Stacks. When control flow data is stored on the same stack as user data (data which attackers can control), it is easier for attackers to modify control flow to execute their own code. Because control data is stored right next to user data, then an attacker can easily reach and modify control data when writing user data such as in the case of a buffer overflow.

A potential solution is to bridge the implementation and abstraction gap by separating the stack into a User Stack, which stores user data such as local variables, arguments, and buffers, and a Control Stack which stores control flow data such as return addresses and saved base pointers.



Another way to approach this issue is to implement a safe stack and an unsafe stack. In this case, the safe stack stores data with fixed sizes that cannot be overwritten, such as arguments and the saved return address and base pointer. The unsafe stack stores things that can be overwritten, such as pointers and buffers. For this solution, we would need multiple stack pointers.



In reality, however, computers support a linear stack in a single address space. A way to mimic a separate control/safe stack would be to use a secret random space in memory that would be difficult for an attacker to find. Even if an attacker overruns a buffer, this secret location containing control/safe data would be secure.

Even still, this approach is not perfect because the unsafe stack may contain function pointers. Thus, an attacker can overwrite the function pointers to point to shell code. Thus, this solution does not successfully separate instructions from data.

4 Memory Writable or Executable, Not Both (W^X)

Since shell code may be executed from the stack by an attacker, it is important to know how to prevent attackers from being able to complete this process. A way to do this is to implement **W^X ("Write XOR eXecute")**, which ensures that memory is not both writeable and executable at the same time. The memory page permission bits may be set to enforce this. Without the protection of this policy, an attacker can complete the process of writing malicious memory then executing it, because those permissions are not explicitly forbidden. W^X ("Write XOR eXecute") is also known as XN ("eXecute Never") or DEP ("Data Execution Prevention").

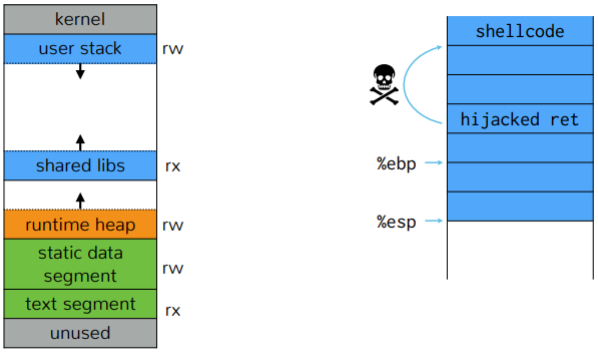
4.1 Tradeoffs

- Easy to deploy: No code changes or recompilation are required to implement this
- Fast: Enforced in hardware rather than software
- Some pages need to be both writeable and executable, such as those that generate code dynamically at runtime (e.g. during just-in-time compilation, or JIT)

Support for this type of defense depends on the hardware. Thus if the system does not have the hardware support for this method, some work around must be applied instead.

4.2 How to Defeat W^X

- An attacker can search the executable for code that can be useful to them, like return into libc. Since instructions are variable length in x86 there is a high likelihood of this being successful.
- An attacker can still write to the return address stored on the stack by overflowing to it. By changing the return address to point to some existing code, this code will be executed after the return.



While this method of defense does not enable the stack to remain executable, code must remain executable. To defeat the W^X defense the attacker can write to the return address on the stack the address of some existing code. This enables the attacker to search the code for what they want. The attacker could call system("/bin/sh") or any other chunk of code that meets their malicious needs. The idea behind this attack is to find or inject some code the attacker wants. Once this is done, all the attacker needs to do is overwrite the stack pointer to point to the code the attacker wants executed.

4.3 How to inject code

- JIT Spraying
  - Spray the heap with shellcode; this is shown in the image below, where the attacker fills the heap with many copies of some code that they want to run.
  - Overflow code pointer to spray the heap; this arms the attacker with the knowledge of the general location of the code from the sprayed heap to run their own code.

Just-In-Time (JIT) compilers produce data that becomes executable code. Therefore, an attacker can make a huge number of requests that fills the heap with shellcode. This type of attack is how most modern buffer overflow attacks work. The following is an example of this [1]:

```
1  var g1 = 0;
2  ...
3  var g7 = 0;
4
5  for (var i=0; i<100000; ++i) {
6          g1 = 50011;       \\ pop ebx; ret;
7          g2 = 50009;       \\ pop ecx; ret;
8          g3 = 12828721;    \\ xor eax, eax; ret;
9          g4 = 12811696;    \\ mov 0x7d, al; ret;
10         g5 = 12833329;    \\ xor edx, edx; ret;
11         g6 = 12781490;    \\ mov 0x7,  dl; ret;
12         g7 = 12812493;    \\ int 0x80; ret;
13 }
```

Once the code fills up the heap, the code continues to run until it lands on the NOP slide. This slides the execution flow to the shellcode that the attacker wants to run.

5 Address Space Layout Randomization

The next and final mitigation we'll look at is Address Space Layout Randomization (ASLR). Throughout the exploits we have described so far, a malicious attacker has used precise locations that they can reroute the control flow to. Doing this allows them the freedom to execute a piece of shellcode or perform a return-into-libc attack. ASLR attempts to make it harder for the attacker to use these precise addresses by randomizing the address of different memory regions.

While ASLR is employed by many modern machines, there are a few downsides to this approach. Firstly, ASLR can be quite intrusive. The compiler, linker, and loader must have access in order to ensure that process layout

is randomized and that programs have been compiled to not have absolute jumps. Also, ASLR can have an overhead attached with it. It not only causes the size of the code to increase, but performance also suffers. It is not all bad news though, as ASLR can mitigate against potential heap-based overflow attacks.

5.1 How to Defeat ASLR

Every region has a random offset, but the layout within the region is fixed. If the attacker is able to learn an address of something within that region, then they can learn the offset. By having knowledge of the offset it can leak any address in the region.

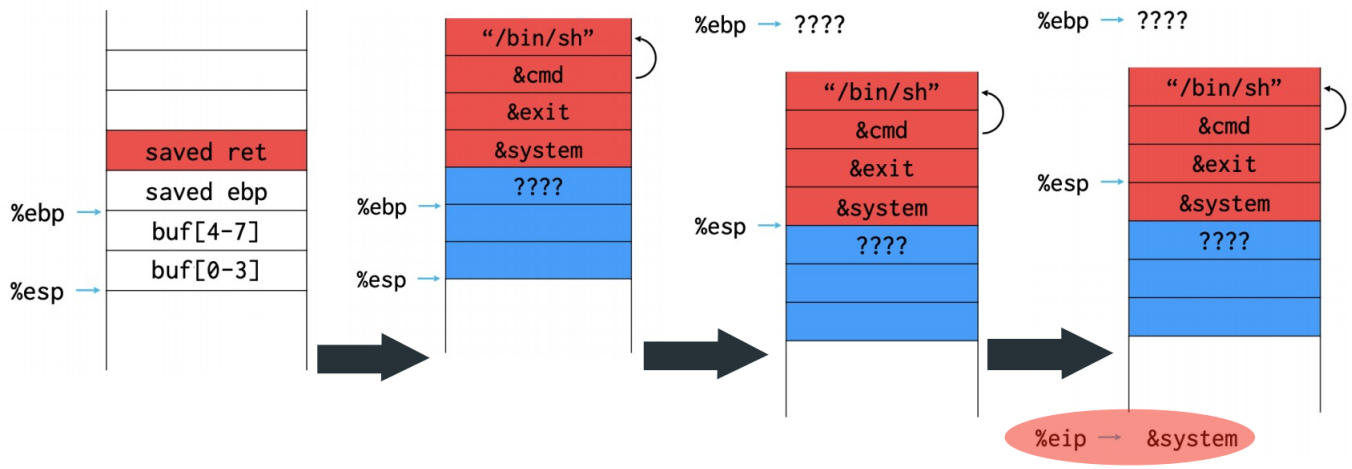
With 32-bit binaries it is possible to brute force 16 bits of randomness. If the process is forking then you can learn the value in the fork and use that to attack the forked value. With 64-bit binaries you can completely fill the heap with garbage value which will allow you to point somewhere arbitrary that may potentially land you in a vulnerable area. This is referred to as heap spraying.

Specific Example of return into libc

At this point, we have discussed how a buffer overflow can be exploited to gain control of the execution of the program along with multiple mitigations. Lastly, we'll take a look at a typical buffer overflow exploitation and will explain step-by-step how it works.

Redirecting Control Flow to system()

This exploit like all buffer overflows, will attempt to gain control of the instruction pointer and thereby control the flow of the program's execution. In the past, execution has been pointed to a function such as foo() which has included an innocuous print statement. Now, the attacker intends to do something much more malicious by opening up a root shell, giving them almost total control. The best way to understand how this will work is by looking at a succession of snapshots of the stack, with each representing the execution state of the program while it is being exploited. Each snapshot will be explained down below.



The first snapshot shows execution before anything has been exploited. Note that there is a buffer of size 8, and with ample opportunity (ie. unsafe function) an attacker could write data past the end of buf and overwrite the function base pointer register and the stored return address. The attacker will overwrite the return address with '&system' but equally important, they also write '&exit', '&cmd', and the string '/bin/sh' onto the stack.

This takes us to the second snapshot which occurs after we have overwritten the return address with '&system'. The value pointed to by the base pointer has been filled with ???? values. Now, the process of returning from

the current function begins. The stack pointer (esp) is set equal to the value of the function base pointer (ebp). ebp is then popped, which subtracts 4 from the stack pointer, setting esp equal to the saved return address that we overwrote with '&system'. While it may seem like a lot to take in, the new values of the registers (ebp and esp) are shown in the third snapshot of the stack.

The stack pointer now points to the saved return address which contains the value '&system'. The next step in the return process is to pop the instruction pointer, which will place the value currently pointed to by the stack pointer (esp) into the instruction pointer register (eip). This change can be viewed in the fourth snapshot.

The value of register eip is '&system'. The register eip determines the address of the next instruction to be executed. The next instruction to be executed is therefore 'system' and the attacker has managed to control the instruction pointer to point to exactly what they would like. At this point, it looks like a normal call to the system. This is shown by the next and final stack snapshot complete with an explanation below it.

The stack on the left-hand side is what a stack usually looks like performing a "normal" `system()` call. On the right is our stack, which is also performing a `system()` call, complete with all of the arguments the attacker wrote onto the stack back in step one. As you can see, the attacker has set up the return address to be equal to '&exit' and arg0 to be 'cmd' which points to '/bin/sh' on the stack. When the right-sided stack's execution completes, this will look like a normal system call and the attacker will have full control of the system with the root shell that is spawned.

References

[1] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. *Proceedings 2015 Network and Distributed System Security Symposium*, 2015.