Memory safety defenses

Myrto Arapinis
School of Informatics
University of Edinburgh

Key techniques against memory safety attacks

1. Use memory-safe languages - checks on buffer bounds are automated by the compiler

2. Apply safe programming practices - when using non-memory safe languages check all the bounds, and validate user input

3. Code hardening - OS and compiler based techniques to defend against BOs
   3.1 Stack canaries
   3.2 Data Execution Protection (DEP) / Write XOR Execute (W^X)
   3.3 Address Space Layout Randomisation (ASLR)

Memory-safe languages

- Memory-safe languages are not subject to memory safety vulnerabilities:
  - Access to memory is well-defined
  - Checks on array bounds and poiner dereferences are automatically included by the compiler
  - Garbage collection takes away from the programmer the error-prone task of managing memory
- Plenty of memory-safe languages: Java, Python, Rust, Go, *etc.*
- Whenever possible in new projects use a memory-safe programming language!

Safe programming practices

- Use safe C libraries - Size-bounded analogues of unsafe libc functions

```
size_t strlcpy(char *destination, const char *source, size_t size);
size_t strlcat(char *destination, const char *source, size_t
size);
char *fgets(char *str, int n, FILE *stream);
...
```

- Check bounds and validate user input

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Create a buffer on the stack
  char buf[256];
  // Only copy as much of the argument as can fit in the buffer
  strcpy(buf, argv[1]);
  // Print the contents of the buffer
  printf(``%s\n'', buf);
  return 1;
  }
```

Safe programming practices

- Use safe C libraries - Size-bounded analogues of unsafe libc functions

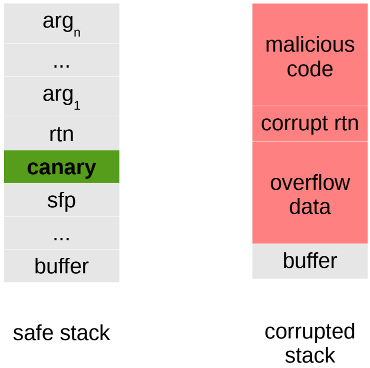
```
size_t strlcpy(char *destination, const char *source, size_t size);
size_t strlcat(char *destination, const char *source, size_t
size);
char *fgets(char *str, int n, FILE *stream);
...
```

- Check bounds and validate user input

```
#include <stdio.h>
int main(int argc, char *argv[]){
  // Create a buffer on the stack
  char buf[256];
  // Only copy as much of the argument as can fit in the buffer
  strlcpy(buf, argv[1], sizeof(buf));
  // Print the contents of the buffer
  printf(``%s\n'', buf);
  return 1;
  }
```

Stack canaries

- Goal: detect a stack buffer overflow before execution of malicious code
- Idea: place trap (the canary) just before the stack return pointer
- The value of the canary needs to be a randomly chosen fresh value for each execution of the program
- To overwrite the return pointer the canary value must also be overwritten
- The canary is checked to make sure it has not changed before returning



Limitations of stack canaries

Stack canaries will detect a BO if

- The attacker does not learn the value of the canary - this could happen through a buffer overread
- The attacker cannot jump over the canary - the assumption is that the attacker has to write consecutively memory from buffer to return address
- The attacker cannot guess the canary value - on 32-bits the attacker might be able to brute force the canary value
- The buffer overrun occurs on the stack - canaries will not detect heap overruns

Take way

Stack canaries make attacks harder but not impossible!

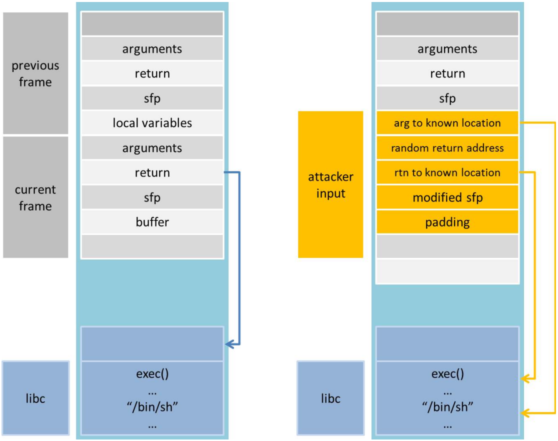
Data Execution Protection (DEP)
Write XOR Execute (W^X)

- Goal: prevent malicious code from being executed.
- Idea: Make regions in memory either executable or writable (but not both)
- The stack and heap will be writable but not executable because they only store data
- The text segment will only be executable and not writable because it only stores code

:-) Even if the attacker manages to put his malicious code on stack or heap, it will never get executed :-)

Limitation of W^X: `return-to-libc` attacks

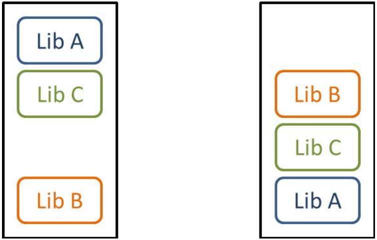
- the attacker does not need to inject any code
- the `libc` library is linked to most C programs
- `libc` provides useful calls for an attacker



Address Space Layout Randomization (ASLR)

- Goal: prevent that attacker from predicting where things are in memory
- Idea: place standard libraries to random locations in memory
  $\longrightarrow$ for each process, `exec()` is situated at a different location
  $\longrightarrow$ the attacker cannot directly point to `exec()`

- Supported by most operating systems (Linux, Windows, MAC OS, Android, iOS, ...)



But ultimately

- Hackers have and will develop more complicated ways of exploiting buffer overflows.
- It all boils down to the programmer.
- The most important preventive measure is: safe programming
- Whenever a program copies user-supplied input into a buffer ensure that the program does not copy more data than the buffer can hold

Take away message

OSes may have features to reduce the risks of BOs, but the best way to guarantee safety is to remove these vulnerabilities from application code.