CE4062/CZ4062
Computer Security

Tutorial 4: Software Vulnerability Defenses

Tianwei Zhang

Q1

Short answers

(a) Why are non-executable stack and heap not enough to defeat buffer overflow attacks?

(b) Distinguish three types of fuzzing techniques?

(c) What is a code reuse attack?

Solution

Short answers

(a) Why are non-executable stack and heap not enough to defeat buffer overflow attacks?

The return address can be overwritten to return to any code already loaded. For instance, the attacker may be able to cause a return into the libc execve() function with "/bin/sh" as an argument.

Solution

Short answers

b) Distinguish three types of fuzzing techniques?

- *Mutation-based fuzzing: randomly perturb the input in a heuristic way to test whether the system will crash*
- *Generation-based fuzzing: generate test cases based on the specification of the input format to test*
- *Coverage-based fuzzing: craft test cases based on the feedback of code coverage*

Solution

Short answers

c) What is a code reuse attack?

Instead of injecting the malicious code into the memory, the attacker can compromise the control flow to jump to the existing library for attacks.

Q2

For string copy operation, `strncpy` is regarded as "safer" than `strcpy`. However, improper use of `strncpy` can also incur vulnerabilities. Please describe the problem in the following program, and what consequences it will cause.

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

int main() {
  char src[] = "geeksforgeeks";

  char dest[8];
  strncpy(dest, src, 8);
  int len = strlen(dest);

  printf("Copied string: %s\n", dest);
  printf("Length of destination string: %d\n", len);

  return 0;
}
```

Solution

Does not automatically add the NULL value to dest if n is less than the length of string src. So it is safer to always add NULL after strncpy.

- In this program, dest does not have a terminator at the end. Then the function `strlen` will keep counting, which can cause segmentation fault.

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

int main() {
  char src[] = "geeksforgeeks";

  char dest[8];
  strncpy(dest, src, 8);
  int len = strlen(dest);

  printf("Copied string: %s\n", dest);
  printf("Length of destination string: %d\n", len);

  return 0;
}
```

Q3

StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It detects memory corruption using a canary, a known value stored in each function's stack frame immediately before the return address.

(a) In some implementations, the canary value is a 64-bit integer that is randomly generated each time the program runs. Explain why this prevents the basic form of buffer-overflow attacks

(b) What is a security drawback to choosing the canary value at compile time instead of at run time?

(c) If the value must be fixed, what will be a good choice?

(d) List an attack which can defeat the StackGuard.

Solution

StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It detects memory corruption using a canary, a known value stored in each function's stack frame immediately before the return address.

a) In some implementations, the canary value is a 64-bit integer that is randomly generated each time the program runs. Explain why this prevents the basic form of buffer-overflow attacks

This guarantees the attacker cannot guess canary correctly every time, and cannot corrupt the stack without changing the canary values.

Solution

StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It detects memory corruption using a canary, a known value stored in each function's stack frame immediately before the return address.

b) What is a security drawback to choosing the canary value at compile time instead of at run time?

After compiling, the canary value will be fixed. The attacker may use brute-force attack to guess the correct value, and then perform the buffer overflow attack with that value.

Solution

StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It detects memory corruption using a canary, a known value stored in each function's stack frame immediately before the return address.

c) If the value must be fixed, what will be a good choice?

Using terminator canary: {\0, newline, linefeed, EOF}. The attacker cannot copy strings beyond the terminator.

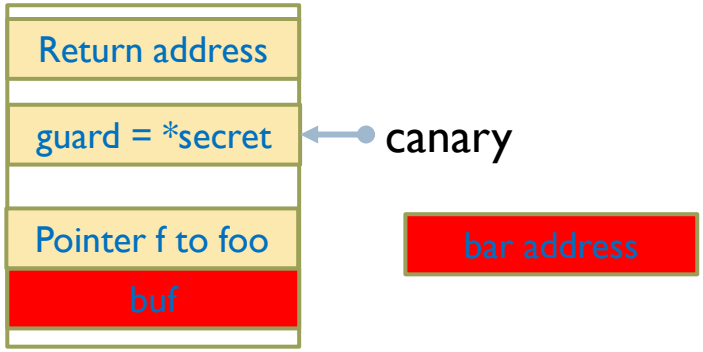
Solution

StackGuard is a mechanism for defending C programs against stack-based buffer overflows. It detects memory corruption using a canary, a known value stored in each function's stack frame immediately before the return address.

d) List an attack which can defeat the StackGuard.

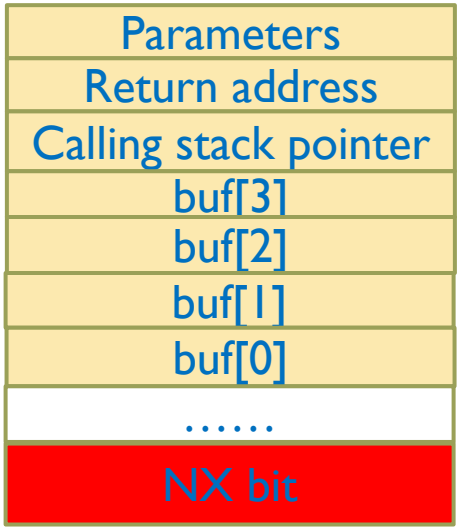
The attacker can overwrite the function pointer instead of the return address

```
void foo () {...}
void bar () {...}
int main() {
    void (*f) () = &foo;
    char buf [16];
    gets(buf);
    f();
}
```



Q4

One possible solution to defeat buffer overflow attacks is to set the stack memory as Non-executable (NX). This is usually achieved by the OS and the paging hardware. However, imagine that a machine does not support the non-executable feature, then we can implement this functionality at the software level. The compiler can allocate each stack frame in a separate page, and associate a software-manipulated NX bit at the bottom of this page, controlling if this page (stack frame) is non-executable. The structure of a stack frame is shown in the Figure below.



Since the NX bit is at the bottom of the memory page, the buffers inside this stack frame is not able to overwrite this bit to make it executable.

Describe a buffer overflow attack that can still overwrite NX bits.

Solution: Overflow from callee frame

Although the buffers inside the frame cannot overwrite the NX bit. But it can call another function, whose buffer can be used by the attacker to overwrite the NX bit. Then the caller's frame will become executable.

```
foo( ) {
  bar( );
}
bar( ) {
  char buf[4];
}
```