Hacking in C

Attacks, part I

Radboud University, Nijmegen, The Netherlands

Spring 2018

A short recap

- Program memory is partitioned into different segments
- Code segment (or `.text`) for (read-only) program code
- `.data` and `.bss` for global and static variables

A short recap

- Program memory is partitioned into different segments
- Code segment (or `.text`) for (read-only) program code
- `.data` and `.bss` for global and static variables
- Stack for local data of functions
  - Grows downwards, i.e., function call decreases stack pointer
  - Also contains return addresses, function arguments, (frame pointer)
  - Managed *automatically*, data is non-persistent
  - Stack overflow: exceeding maximum stack size (e.g., massive recursion)

A short recap

- Program memory is partitioned into different segments
- Code segment (or `.text`) for (read-only) program code
- `.data` and `.bss` for global and static variables
- Stack for local data of functions
  - Grows downwards, i.e., function call decreases stack pointer
  - Also contains return addresses, function arguments, (frame pointer)
  - Managed *automatically*, data is non-persistent
  - Stack overflow: exceeding maximum stack size (e.g., massive recursion)
- Heap for persistent or large data
  - Request heap space with `malloc`
  - Resize requested memory with `realloc`
  - Always check whether returned pointer is `NULL`!
  - Free heap space using `free`
  - Heap is managed by the programmer
  - Many possible problems: danging pointers, double-free, memory leaks. . .

`calloc`

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized

`calloc`

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized
- Alternative: use `calloc`:

```
void *calloc(size_t nitems, size_t size)
```

- Request space for `nitems` elements of size `size` each
- Memory space is initialized to zero

calloc

- Remember that data on the stack is not initialized
- Global variables are initialized
- Memory space allocated with `malloc` is *not* initialized
- Alternative: use `calloc`:

```
void *calloc(size_t nitems, size_t size)
```

- Request space for `nitems` elements of size `size` each
- Memory space is initialized to zero
- Example usage:

```
int *p = calloc(1000, sizeof(int));
if(p == NULL) exit(-1);
```

- Request space for 1000 integers

`malloc` vs. `calloc`

- Aside from initialization, any difference between
  - `int *p = malloc(nelems*sizeof(int));` and
  - `int *p = calloc(nelems,sizeof(int));`?

`malloc` vs. `calloc`

- Aside from initialization, any difference between
  - `int *p = malloc(nelems*sizeof(int));` and
  - `int *p = calloc(nelems,sizeof(int));`?
- Multiplication `nelems*sizeof(int)` can overflow!
- Result: successful allocation, but of *much less* memory!

`malloc` vs. `calloc`

- Aside from initialization, any difference between
  - `int *p = malloc(nelems*sizeof(int));` and
  - `int *p = calloc(nelems,sizeof(int));`?
- Multiplication `nelems*sizeof(int)` can overflow!
- Result: successful allocation, but of *much less* memory!
- Another difference:
  - `malloc` doesn't guarantee you that you can *use* the memory you requested
  - Linux optimistically grants you the memory
  - Later *access* to this memory may still fail
  - `calloc` gives you memory that is actually "backed" by the OS

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

- Question: How does `free` know, how much memory belongs to a pointer?

Heap management

- Remember `free`?:

  ```
  int *p = malloc(1000*sizeof(int));
  if(p == NULL) exit(-1);
  ...
  free(p);
  ```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap
- One solution: maintain a table of all malloc'ed addresses and space

Heap management

- Remember `free`?:

```
int *p = malloc(1000*sizeof(int));
if(p == NULL) exit(-1);
...
free(p);
```

- Question: How does `free` know, how much memory belongs to a pointer?
- Answer: `malloc` needs to write this information somewhere
- Obvious location: the heap
- One solution: maintain a table of all malloc'ed addresses and space
- Other solution: write information just before the pointer

Reading and manipulating data

- Situation so far: program itself may do “weird” things in its own memory space
- “Weird” things means: **undefined behavior**
- Easiest case: program crashes (segmentation fault)
- More scary case: leak data (out of bounds read)

Reading and manipulating data

- Situation so far: program itself may do “weird” things in its own memory space
- “Weird” things means: **undefined behavior**
- Easiest case: program crashes (segmentation fault)
- More scary case: leak data (out of bounds read)
- Question now: How can we use *inputs* to the program to
  - Leak data

Reading and manipulating data

- Situation so far: program itself may do “weird” things in its own memory space
- “Weird” things means: **undefined behavior**
- Easiest case: program crashes (segmentation fault)
- More scary case: leak data (out of bounds read)
- Question now: How can we use *inputs* to the program to
  - Leak data
  - Manipulate program’s data

Reading and manipulating data

- Situation so far: program itself may do "weird" things in its own memory space
- "Weird" things means: **undefined behavior**
- Easiest case: program crashes (segmentation fault)
- More scary case: leak data (out of bounds read)
- Question now: How can we use *inputs* to the program to
  - Leak data
  - Manipulate program's data
  - Have the program do something completely different
  - Take full control over what the program does

Reading and manipulating data

- Situation so far: program itself may do “weird” things in its own memory space
- “Weird” things means: **undefined behavior**
- Easiest case: program crashes (segmentation fault)
- More scary case: leak data (out of bounds read)
- Question now: How can we use *inputs* to the program to
  - Leak data
  - Manipulate program’s data
  - Have the program do something completely different
  - Take full control over what the program does
- Remember: All of this is allowed by the C specification!

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets
  - E-mail client, receiving and parsing (html) e-mails

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets
  - E-mail client, receiving and parsing (html) e-mails
  - Web browser parsing html and executing JavaScript.

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets
  - E-mail client, receiving and parsing (html) e-mails
  - Web browser parsing html and executing JavaScript.
  - Messaging (chat) programs receiving messages

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets
  - E-mail client, receiving and parsing (html) e-mails
  - Web browser parsing html and executing JavaScript.
  - Messaging (chat) programs receiving messages
  - Any program you use to open e-mail attachments

Attack scenario

- Attacker does not actually run the program
- Attacker only provides *input* to the program
- Examples of software accepting attacker input:
  - Internet server (web, mail, etc.) receiving packets
  - E-mail client, receiving and parsing (html) e-mails
  - Web browser parsing html and executing JavaScript.
  - Messaging (chat) programs receiving messages
  - Any program you use to open e-mail attachments
- **Is there any program that only receives trustworthy input?**

Format-string attacks

Format-string attacks

- Format-string attacks were first described in 1999
- This is after 25 years of people programming in C!
- Allow an attacker to
  - read data from the stack (and heap)
  - manipulate data in memory

Format-string attacks

- Format-string attacks were first described in 1999
- This is after 25 years of people programming in C!
- Allow an attacker to
  - read data from the stack (and heap)
  - manipulate data in memory
- Vulnerable code (typically) fairly easy to spot
- These attacks *should* be history by now
- Still, great first example for malicious-input attacks

A simple `main`

```
#include<stdio.h>

extern long f(long *x);

int main(int argc, char* argv[])
  long pincode = 1234;
  printf (argv [1]);
  return f(&pincode);
}
```

- The program prints the first command line argument
- The call to `f()` is so that gcc doesn't optimize `pincode` away

argc and argv

- Two options for main arguments:
  - (void)
  - (int argc, char **argv) or (int argc, char *argv[])
- char **argv and char *argv[] are equivalent

argc and argv

- Two options for `main` arguments:
  - `(void)`
  - `(int argc, char **argv)` or `(int argc, char *argv[])`
- `char **argv` and `char *argv[]` are equivalent
- `char **` is a pointer to a pointer to `char`
- Can also see it as an array of strings
- Contains the command-line arguments:
  - `argv[0]` is the name of the program
  - `argv[1]` is the first argument
  - `argv[2]` is the second argument
  - ...

argc and argv

- Two options for main arguments:
  - (void)
  - (int argc, char **argv) or (int argc, char *argv[])
- char **argv and char *argv[] are equivalent
- char ** is a pointer to a pointer to char
- Can also see it as an array of strings
- Contains the command-line arguments:
  - argv[0] is the name of the program
  - argv[1] is the first argument
  - argv[2] is the second argument
  - ...
- argc contains the length of the argv array
- Without any command-line argument, argc == 1

Arguments passing, the “easy” way

- Let's call our program with `./formatstring "%x"`
- What does the program print?

Arguments passing, the “easy” way

- Let’s call our program with `./formatstring "%x"`
- What does the program print?
- Answer: It depends ;-)
- `printf` has variable number of arguments
- For each % in the first argument, expect one more argument
- (To print a percent sign, use %% in the first printf argument)

Arguments passing, the “easy” way

- Let’s call our program with `./formatstring "%x"`
- What does the program print?
- Answer: It depends ;-)
- `printf` has variable number of arguments
- For each % in the first argument, expect one more argument
- (To print a percent sign, use %% in the first printf argument)
- `printf` finds one `%x`, looks for second argument
- Traditionally (x86) arguments are passed through the stack
- On x86, we would get the top 4 bytes of the stack

Arguments passing on AMD64

- Passing arguments over the stack is expensive
- Caller needs to store, callee needs to load
- Advantage: doesn't need registers (only 8 of those on x86!)

Arguments passing on AMD64

- Passing arguments over the stack is expensive
- Caller needs to store, callee needs to load
- Advantage: doesn't need registers (only 8 of those on x86!)
- AMD64 extended register size from 32 to 64 bits
- AMD64 also introduced 8 more registers (`r8`,. . . ,`r15`)

Arguments passing on AMD64

- Passing arguments over the stack is expensive
- Caller needs to store, callee needs to load
- Advantage: doesn't need registers (only 8 of those on x86!)
- AMD64 extended register size from 32 to 64 bits
- AMD64 also introduced 8 more registers (`r8`,...,`r15`)
- Major change in function-call ABI: pass up to 6 arguments through registers:
  - First argument passed through `rdi`
  - Second argument passed through `rsi`
  - ...
- `printf` will look for second argument in `rsi`
- Output (on AMD64) is the content of `rsi`

Arguments passing on AMD64

- Passing arguments over the stack is expensive
- Caller needs to store, callee needs to load
- Advantage: doesn't need registers (only 8 of those on x86!)
- AMD64 extended register size from 32 to 64 bits
- AMD64 also introduced 8 more registers (`r8`,...,`r15`)
- Major change in function-call ABI: pass up to 6 arguments through registers:
  - First argument passed through `rdi`
  - Second argument passed through `rsi`
  - ...
- `printf` will look for second argument in `rsi`
- Output (on AMD64) is the content of `rsi`
- What happens if we run
  `./formatstring "%p %p %p %p %p %p %p %p %p"`?

Format strings

- A *format string* is a string containing control parameters
- Control parameters start with `%`
- Functions like `printf` (or `snprintf`) interpret those control parameters
- Look in additional arguments for values to replace them
- Short recap:
  - `%d` prints integer in decimal
  - `%x` prints integer in hexadecimal
  - `%p` prints a pointer
  - `%s` prints a string

Leaking data with format strings

- Exploit code of the form `printf(str)`
- Assume that `str` is controlled by attacker

Leaking data with format strings

- Exploit code of the form `printf(str)`
- Assume that `str` is controlled by attacker
- Choosing `str` to contain many `%p` prints
  - first the values of `rsi`, `rdx`, `rax`, `r8`, `r9`
  - then the content of the stack

Leaking data with format strings

- Exploit code of the form `printf(str)`
- Assume that `str` is controlled by attacker
- Choosing `str` to contain many `%p` prints
  - first the values of `rsi`, `rdx`, `rax`, `r8`, `r9`
  - then the content of the stack
- What happens if we use `str == "%s"`?

Leaking data with format strings

- Exploit code of the form `printf(str)`
- Assume that `str` is controlled by attacker
- Choosing `str` to contain many `%p` prints
  - first the values of `rsi`, `rdx`, `rax`, `r8`, `r9`
  - then the content of the stack
- What happens if we use `str == "%s"`?
- Print memory from address in `rdi` up to first zero byte
- Use multiple `%s` to print memory at multiple locations

Corrupting data with format strings

- So far we can use format-string attacks “only” to *read* data
- Consider the following code program:

```
int main(int argc, char *argv[])
{
  int len;
  printf("How long is %s?\n%n", argv[1], &len);
  printf("%d\n", len-14);
  return 0;
}
```

Corrupting data with format strings

- So far we can use format-string attacks “only” to *read* data
- Consider the following code program:

```
int main(int argc, char *argv[])
{
  int len;
  printf("How long is %s?\n%n", argv[1], &len);
  printf("%d\n", len-14);
  return 0;
}
```

- The `%n` control parameter causes `printf` to *write* data
- Write number of characters printed so far to `len`

Corrupting data with format strings

- So far we can use format-string attacks “only” to *read* data
- Consider the following code program:

```
int main(int argc, char *argv[])
{
  int len;
  printf("How long is %s?\n%n", argv[1], &len);
  printf("%d\n", len-14);
  return 0;
}
```

- The `%n` control parameter causes `printf` to *write* data
- Write number of characters printed so far to `len`
- What happens if we feed `"Hello World!%n"` to:

```
int main(int argc, char* argv[])
  printf (argv [1]);
}
```

Corrupting data with format strings

- So far we can use format-string attacks "only" to *read* data
- Consider the following code program:

```
int main(int argc, char *argv[])
{
  int len;
  printf("How long is %s?\n%n", argv[1], &len);
  printf("%d\n", len-14);
  return 0;
}
```

- The `%n` control parameter causes `printf` to *write* data
- Write number of characters printed so far to `len`
- What happens if we feed `"Hello World!%n"` to:

```
int main(int argc, char* argv[])
  printf (argv [1]);
}
```

- Answer (on AMD64): Write 12 to the address in `rsi`
- Can choose arbitrary values by feeding in longer strings

Evil formatstrings (summary)

- Use `%p`, `%x` to print data in (argument) registers and the stack
- Use `%n` to write data to addresses you *don't* control

Evil formatstrings (summary)

- Use `%p`, `%x` to print data in (argument) registers and the stack
- Use `%n` to write data to addresses you *don't* control
- How about writing to addresses you *do* control?
- Assumption is that attacker controls the format string
- Typically:
  - format string sits somewhere on the stack
  - Same location that will be used by `printf` for arguments

Evil formatstrings (summary)

- Use `%p`, `%x` to print data in (argument) registers and the stack
- Use `%n` to write data to addresses you *don't* control
- How about writing to addresses you *do* control?
- Assumption is that attacker controls the format string
- Typically:
  - format string sits somewhere on the stack
  - Same location that will be used by `printf` for arguments
- Can *choose* address to write to with `%n`
- Depends on where the actual format-string is sitting in memory

The first format-string attack

- Discovered by Tymm Twillman when auditing ProFTPD (1999)
- First ftp to vulnerabable host, login (anonymous)

  ```
  ftp> ls aaaXXXX%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u
       %u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%653300u%n
  ```

- replace the X's with the characters with ASCII values 0xdc,0x4f,0x07,0x08 consecutively
- See http://seclists.org/bugtraq/1999/Sep/328

The first format-string attack

```
ftp> ls aaaXXXX%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u
     %u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%u%653300u%n
```

"The a's at the beginning are just for alignment, the `%u`*'s to skip bytes in the stack, the* `%653300u` *is to increment the # of bytes that have been "output", and the* `%n` *stores that value (whose LSBs have now flipped over to 0) to the location pointed to by the current "argument" – which just happens to point right after the a's in this string. The bytes that replace the X's are the address where proftpd keeps the current user ID..."*

Getting rid of format-string attacks

- Vulnerable pattern:

```
printf(str);
```

- Safe pattern:

```
printf("%s", str);
```

- Attacker does not control the first argument to `printf` anymore
- Control parameters are only in the first argument

Getting rid of format-string attacks

- Vulnerable pattern:

```
printf(str);
```

- Safe pattern:

```
printf("%s", str);
```

- Attacker does not control the first argument to `printf` anymore
- Control parameters are only in the first argument
- Compilers find such patterns and warn about them
- Need compiler flags (e.g., `-Wall`)

Getting rid of format-string attacks

- Vulnerable pattern:

```
printf(str);
```

- Safe pattern:

```
printf("%s", str);
```

- Attacker does not control the first argument to `printf` anymore
- Control parameters are only in the first argument
- Compilers find such patterns and warn about them
- Need compiler flags (e.g., `-Wall`)
- Much more subtle if format string is not known at compile time
- Could ask a user: how would you like your output formatted
- Static (compile-time) analysis has no chance here
- Need to carefully validate user input!

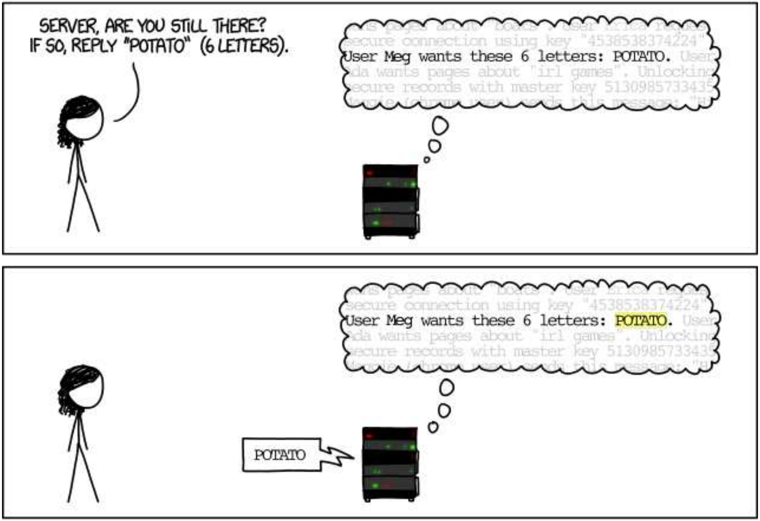
Buffer Overflows

No bounds checking – what could go wrong?

- April 7, 2014, OpenSSL discloses "Heartbleed" bug
- Heartbleed allows remote attacker to read out OpenSSL memory
- Content typically includes cryptographic keys, passwords, etc.
- Bug was in OpenSSL for more than 3 years
- Introduced on December 31, 2010
- First bug with a logo
- Major media coverage
- Initiated major changes in OpenSSL

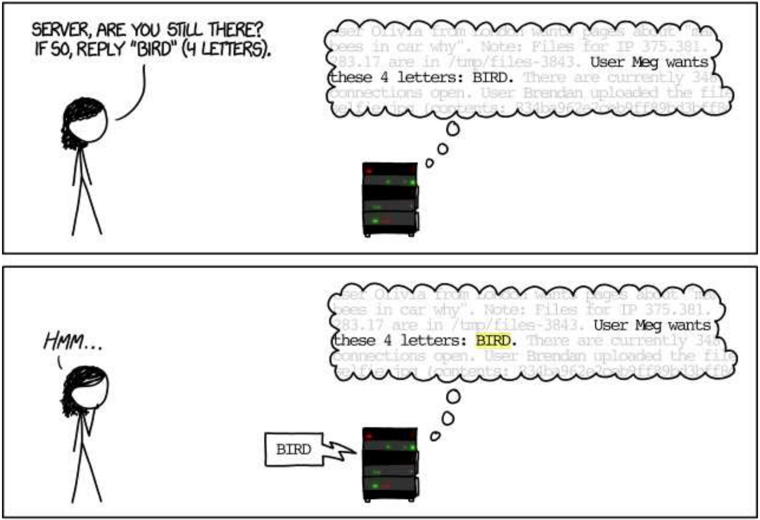
Underlying problem: Out of bounds array access in OpenSSL

How Heartbleed works



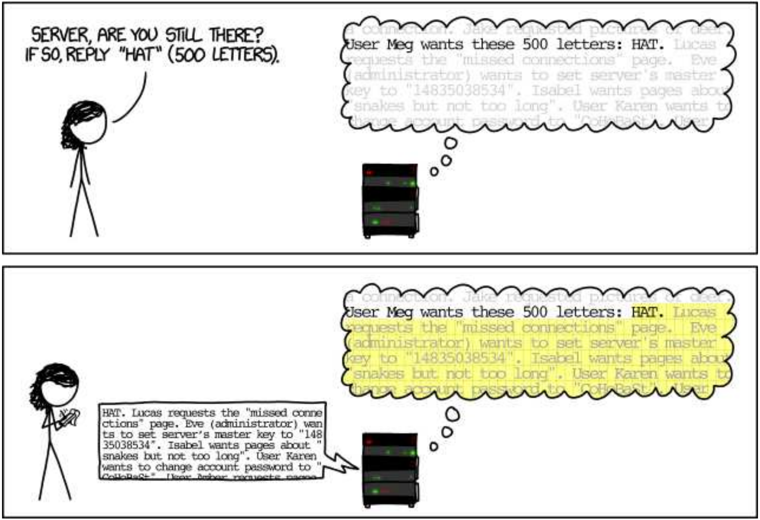
https://xkcd.com/1354/

How Heartbleed works



https://xkcd.com/1354/

How Heartbleed works



https://xkcd.com/1354/

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset + packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get “chopped” into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get “chopped” into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset + packet size must not exceed 65535
  - . . . but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535
  - Overlong IP packet will overflow this buffer

Over*writing* data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get “chopped” into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset $+$ packet size must not exceed 65535
  - … but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535
  - Overlong IP packet will overflow this buffer
- This bug was present in UNIX, Linux, Windows, Mac, routers, printers …
- Trivially easy to exploit with some implementations of `ping`:

  ```
  ping -s 65510 target
  ```

Overwriting data on the stack

- Imagine you can send *emph* one packet to crash a server
- Classic example: **ping of death** (mid 90s)
- Idea is the following:
  - IP packets are limited to a length of 65535 bytes
  - IP packets get "chopped" into fragments for transportation through, e.g., Ethernet
  - IP header has a fragment offset
  - Fragment offset + packet size must not exceed 65535
  - ... but it can
  - With fragmentation, it is possible to send IP packets of size > 65535
  - Receiving host will assemble the fragments into a buffer of size 65535
  - Overlong IP packet will overflow this buffer
- This bug was present in UNIX, Linux, Windows, Mac, routers, printers ...
- Trivially easy to exploit with some implementations of `ping`:

  ```
  ping -s 65510 target
  ```

- Fix by checking `offset + packet_size <= 65525`

The return of the ping of death

- CVE-2013-3183: IPv6 ping of death against Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows Server 2012, and Windows RT

The return of the ping of death

- CVE-2013-3183: IPv6 ping of death against Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows Server 2012, and Windows RT
- CVE-2016-1409: IPv6 ping of death against Cisco's IOS, IOS XR, IOS XE, and NX-OS software