Secure by default
Anti-exploit techniques and hardenings in SUSE products

Johannes Segitz
SUSE Security Team

2019-04-02/04

Who am I?

Johannes Segitz, security engineer (Nuremberg, Germany)

- Code review
- Product pentesting

Who am I?

Johannes Segitz, security engineer (Nuremberg, Germany)

- Code review
- Product pentesting

First time in Nashville, great city (but I'll have to go on a diet after this week)

Outline

Buffer overflows and protections:

- Stack canaries
- Fortify source
- Address space layout randomization
- No-execute memory (NX, W^X)
- Stack clash protection
- RELRO

Outline

Buffer overflows and protections:

- Stack canaries
- Fortify source
- Address space layout randomization
- No-execute memory (NX, W^X)
- Stack clash protection
- RELRO

Used by SUSE products, there are other protection mechanisms out there

Outline

Requires some C and assembler background, but we'll explain most on the fly

Outline

Requires some C and assembler background, but we'll explain most on the fly

This is short overview of what we're doing.

General mechanism

We're talking here about **stack** based buffer overflows and counter meassures

General mechanism

We're talking here about **stack** based buffer overflows and counter meassures

A problem in languages in which you manage your own memory (primary example is C)

General mechanism

We're talking here about **stack** based buffer overflows and counter meassures

A problem in languages in which you manage your own memory (primary example is C)

Really simple example:

```c
#include <string.h>

int main(int argc, char **argv) {
  char buffer[20];

  strcpy(buffer, argv[1]);

  return EXIT_SUCCESS;
}
```

General mechanism

The problem is that for a given buffer size too much data is placed in there

General mechanism

The problem is that for a given buffer size too much data is placed in there

Usually a size check is just missing

General mechanism

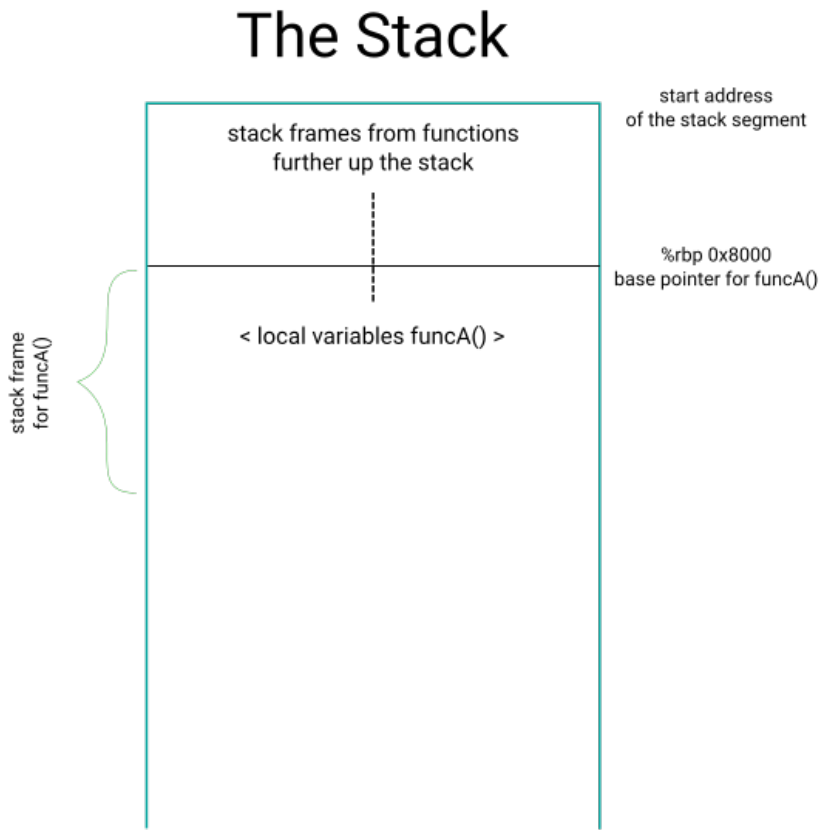
The problem is that for a given buffer size too much data is placed in there

Usually a size check is just missing

Sometimes the check is there but faulty or can be circumvented (think integer overflows)

Why is this a problem?

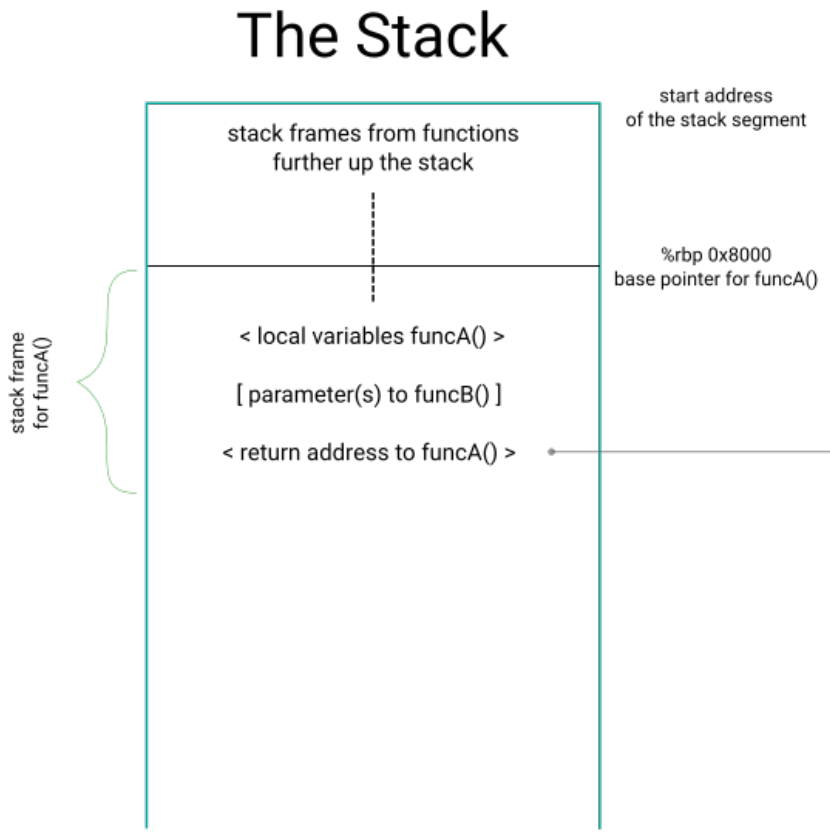
Because in data of the application and control information about execution is mixed




```c
/*
 * some simple, fuzzy code for explaining
 * the stack frame setup
 *
 * hex addresses are just rounded samples
 * for better readability
 */

void funcB(uint32_t num)
{
    uint64_t local_var;
    register uint32_t index;

    /* some more code */
}

void funcA()
{
    /* some local vars */
    funcB(some_num);

    /* some more code */
}
```

Why is this a problem?

Part of the control information (saved instruction pointer RIP/EIP) is the address where execution will continue after the current function




```
/*
 * some simple, fuzzy code for explaining
 * the stack frame setup
 *
 * hex addresses are just rounded samples
 * for better readability
 */

void funcB(uint32_t num)
{
    uint64_t local_var;
    register uint32_t index;

    /* some more code */
}

void funcA()
{
    /* some local vars */
    funcB(some_num);

    /* some more code */
}
```

Why is this a problem?

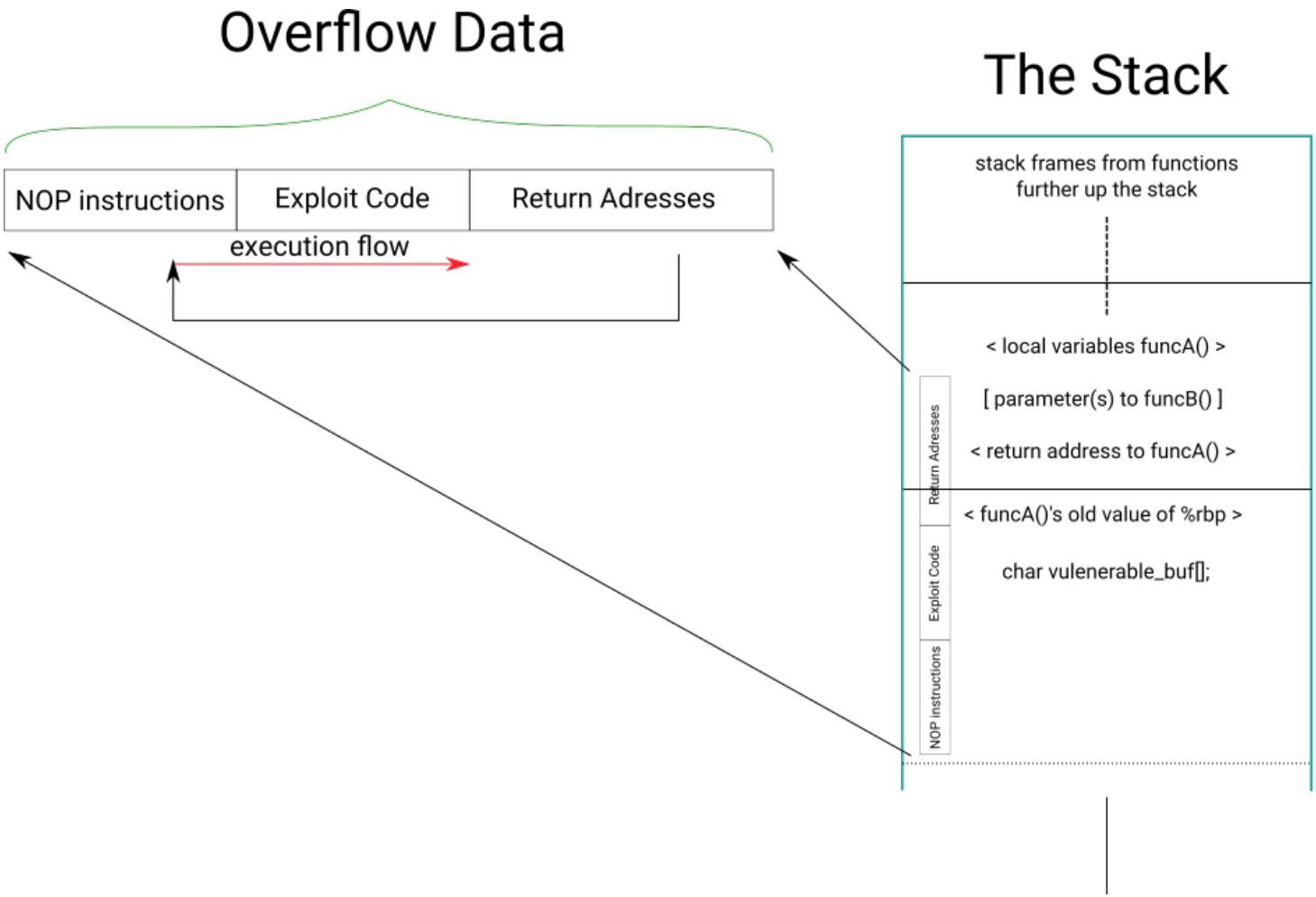
If a buffer overflow happens this control information can be overwritten

Why is this a problem?

If a buffer overflow happens this control information can be overwritten

If this is done carefully arbitrary code can be executed

Why is this a problem?



Other overwrites

Not only saved RIP/EIP can be highjacked. Think of

- Function pointers
- Exceptions handlers
- Other application specific data (`is_admin` flag ...)

Other overwrites

Not only saved RIP/EIP can be highjacked. Think of

- Function pointers
- Exceptions handlers
- Other application specific data (`is_admin` flag ...)

So what can be done against these problems?

Other overwrites

Not only saved RIP/EIP can be highjacked. Think of

- Function pointers
- Exceptions handlers
- Other application specific data (`is_admin` flag ...)

So what can be done against these problems?

Just use Java for everything. Done! We're safe ;)

Simple 32 bit exploitation

```
#include <unistd.h>

void vulnerable( void ) {
  char buffer[256];

  read(0, buffer, 512);

  return;
}

int main(int argc, char **argv) {
  vulnerable();

  return EXIT_SUCCESS;
}
```

Demo time

Mitigations: Stack canaries

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Before returning check if canary is still valid

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Before returning check if canary is still valid

Types:

- Terminator canaries: NULL, CR, LF, and -1

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Before returning check if canary is still valid

Types:

- Terminator canaries: NULL, CR, LF, and -1
- Random canaries

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Before returning check if canary is still valid

Types:

- Terminator canaries: NULL, CR, LF, and -1
- Random canaries
- Random XOR canaries

Mitigations: Stack canaries

General idea: Compiler generates extra code that puts a *canary* value at predefined locations within a stack frame

Before returning check if canary is still valid

Types:

- Terminator canaries: NULL, CR, LF, and -1
- Random canaries
- Random XOR canaries

Enabled since SUSE Linux Enterprise Server 10

Mitigations: Stack canaries

Four variants in gcc:

- `-fstack-protector`: code only for functions that put ≥ 8 bytes buffers on the stack

Mitigations: Stack canaries

Four variants in gcc:

- `-fstack-protector`: code only for functions that put ≥ 8 bytes buffers on the stack
- `-fstack-protector-strong`: additional criteria

Mitigations: Stack canaries

Four variants in gcc:

- `-fstack-protector`: code only for functions that put ≥ 8 bytes buffers on the stack
- `-fstack-protector-strong`: additional criteria
- `-fstack-protector-all`: extra code for each and every function

Mitigations: Stack canaries

Four variants in gcc:

- `-fstack-protector`: code only for functions that put ≥ 8 bytes buffers on the stack
- `-fstack-protector-strong`: additional criteria
- `-fstack-protector-all`: extra code for each and every function
- `-fstack-protector-explicit`: extra code every function annotated with `stack_protect`

Mitigations: Stack canaries

Short reminder of the example code:

```
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[20];

  strcpy(buffer, argv[1]);

  return EXIT_SUCCESS;
}
```

Mitigations: Stack canaries

Original code:

```
00000000000006b0 <main>:
 6b0:   55                      push   rbp
 6b1:   48 89 e5                mov    rbp,rsp
 6b4:   48 83 ec 30             sub    rsp,0x30
 6b8:   89 7d dc                mov    DWORD PTR [rbp-0x24],edi
 6bb:   48 89 75 d0             mov    QWORD PTR [rbp-0x30],rsi
 6bf:   48 8b 45 d0             mov    rax,QWORD PTR [rbp-0x30]
 6c3:   48 83 c0 08             add    rax,0x8
 6c7:   48 8b 10                mov    rdx,QWORD PTR [rax]
 6ca:   48 8d 45 e0             lea    rax,[rbp-0x20]
 6ce:   48 89 d6                mov    rsi,rdx
 6d1:   48 89 c7                mov    rdi,rax
 6d4:   e8 87 fe ff ff          call   560 <strcpy@plt>
 6d9:   b8 00 00 00 00          mov    eax,0x0
 6de:   c9                      leave
 6df:   c3                      ret
```

Mitigations: Stack canaries

Protected code:

```
0000000000000720 <main>:
 720:   55                          push   rbp
 721:   48 89 e5                    mov    rbp,rsp
 724:   48 83 ec 30                 sub    rsp,0x30
 728:   89 7d dc                    mov    DWORD PTR [rbp-0x24],edi
 72b:   48 89 75 d0                 mov    QWORD PTR [rbp-0x30],rsi
 72f:   64 48 8b 04 25 28 00        mov    rax,QWORD PTR fs:0x28
 736:   00 00
 738:   48 89 45 f8                 mov    QWORD PTR [rbp-0x8],rax
 73c:   31 c0                       xor    eax,eax
 73e:   48 8b 45 d0                 mov    rax,QWORD PTR [rbp-0x30]
 742:   48 83 c0 08                 add    rax,0x8
 746:   48 8b 10                    mov    rdx,QWORD PTR [rax]
 749:   48 8d 45 e0                 lea    rax,[rbp-0x20]
 74d:   48 89 d6                    mov    rsi,rdx
 750:   48 89 c7                    mov    rdi,rax
 753:   e8 68 fe ff ff              call   5c0 <strcpy@plt>
 758:   b8 00 00 00 00              mov    eax,0x0
 75d:   48 8b 4d f8                 mov    rcx,QWORD PTR [rbp-0x8]
 761:   64 48 33 0c 25 28 00        xor    rcx,QWORD PTR fs:0x28
 768:   00 00
 76a:   74 05                       je     771 <main+0x51>
 76c:   e8 5f fe ff ff              call   5d0 <__stack_chk_fail@plt>
 771:   c9                          leave
 772:   c3                          ret
```

Mitigations: Stack canaries

Protected code:

```
0000000000000720 <main>:
 720:   55                          push   rbp
 721:   48 89 e5                    mov    rbp,rsp
 724:   48 83 ec 30                 sub    rsp,0x30
 728:   89 7d dc                    mov    DWORD PTR [rbp-0x24],edi
 72b:   48 89 75 d0                 mov    QWORD PTR [rbp-0x30],rsi
 72f:   64 48 8b 04 25 28 00        mov    rax,QWORD PTR fs:0x28
 736:   00 00
 738:   48 89 45 f8                 mov    QWORD PTR [rbp-0x8],rax
 73c:   31 c0                       xor    eax,eax
 73e:   48 8b 45 d0                 mov    rax,QWORD PTR [rbp-0x30]
 742:   48 83 c0 08                 add    rax,0x8
 746:   48 8b 10                    mov    rdx,QWORD PTR [rax]
 749:   48 8d 45 e0                 lea    rax,[rbp-0x20]
 74d:   48 89 d6                    mov    rsi,rdx
 750:   48 89 c7                    mov    rdi,rax
 753:   e8 68 fe ff ff              call   5c0 <strcpy@plt>
 758:   b8 00 00 00 00              mov    eax,0x0
 75d:   48 8b 4d f8                 mov    rcx,QWORD PTR [rbp-0x8]
 761:   64 48 33 0c 25 28 00        xor    rcx,QWORD PTR fs:0x28
 768:   00 00
 76a:   74 05                       je     771 <main+0x51>
 76c:   e8 5f fe ff ff              call   5d0 <__stack_chk_fail@plt>
 771:   c9                          leave
 772:   c3                          ret
```

Mitigations: Stack canaries

Protected code:

```
0000000000000720 <main>:
 720:   55                          push   rbp
 721:   48 89 e5                    mov    rbp,rsp
 724:   48 83 ec 30                 sub    rsp,0x30
 728:   89 7d dc                    mov    DWORD PTR [rbp-0x24],edi
 72b:   48 89 75 d0                 mov    QWORD PTR [rbp-0x30],rsi
 72f:   64 48 8b 04 25 28 00        mov    rax,QWORD PTR fs:0x28
 736:   00 00
 738:   48 89 45 f8                 mov    QWORD PTR [rbp-0x8],rax
 73c:   31 c0                       xor    eax,eax
 73e:   48 8b 45 d0                 mov    rax,QWORD PTR [rbp-0x30]
 742:   48 83 c0 08                 add    rax,0x8
 746:   48 8b 10                    mov    rdx,QWORD PTR [rax]
 749:   48 8d 45 e0                 lea    rax,[rbp-0x20]
 74d:   48 89 d6                    mov    rsi,rdx
 750:   48 89 c7                    mov    rdi,rax
 753:   e8 68 fe ff ff              call   5c0 <strcpy@plt>
 758:   b8 00 00 00 00              mov    eax,0x0
 75d:   48 8b 4d f8                 mov    rcx,QWORD PTR [rbp-0x8]
 761:   64 48 33 0c 25 28 00        xor    rcx,QWORD PTR fs:0x28
 768:   00 00
 76a:   74 05                       je     771 <main+0x51>
 76c:   e8 5f fe ff ff              call   5d0 <__stack_chk_fail@plt>
 771:   c9                          leave
 772:   c3                          ret
```

Demo time

Limitations of stack canaries

Limitations:

Limitations of stack canaries

Limitations:

- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk

Limitations of stack canaries

Limitations:

- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk
- Does not protect against generic write primitives

Limitations of stack canaries

Limitations:

- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk
- Does not protect against generic write primitives
- Can be circumvented with exeption handlers

Limitations of stack canaries

Limitations:

- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk
- Does not protect against generic write primitives
- Can be circumvented with exeption handlers
- Chain buffer overflow with information leak

Limitations of stack canaries

Limitations:

- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk
- Does not protect against generic write primitives
- Can be circumvented with exeption handlers
- Chain buffer overflow with information leak
- No protection for inlined functions

Limitations of stack canaries

Limitations:

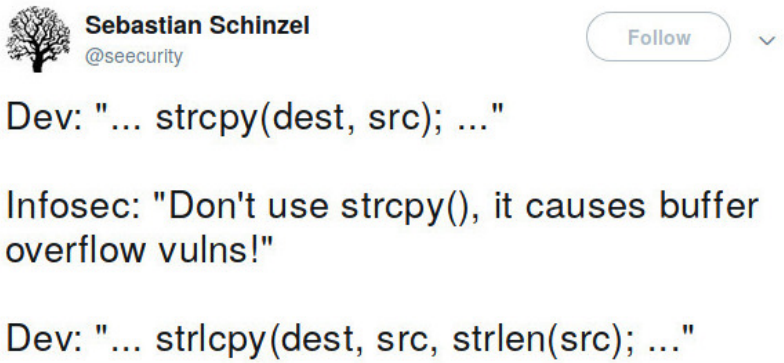
- Does not protect data *before* the canary (especially function pointers). Some implementations reorder variables to minimize this risk
- Does not protect against generic write primitives
- Can be circumvented with exeption handlers
- Chain buffer overflow with information leak
- No protection for inlined functions
- Can be used to cause DoS

Mitigations: Fortify source

Transparently fix *insecure* functions to prevent buffer overflows (memcpy, memset, strcpy, ...).

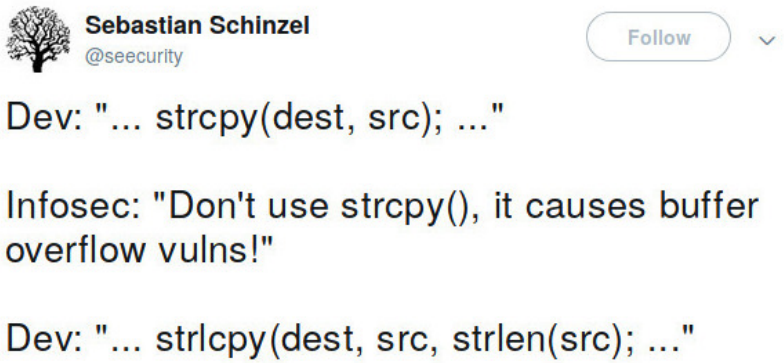
Mitigations: Fortify source

Transparently fix *insecure* functions to prevent buffer overflows (memcpy, memset, strcpy, ...).



Mitigations: Fortify source

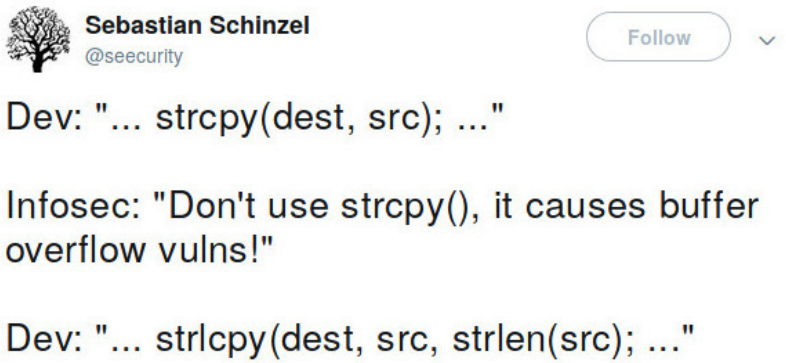
Transparently fix *insecure* functions to prevent buffer overflows (memcpy, memset, strcpy, ...).



What is checked: For statically sized buffers the compiler can check calls to certain functions.

Mitigations: Fortify source

Transparently fix *insecure* functions to prevent buffer overflows (memcpy, memset, strcpy, ...).

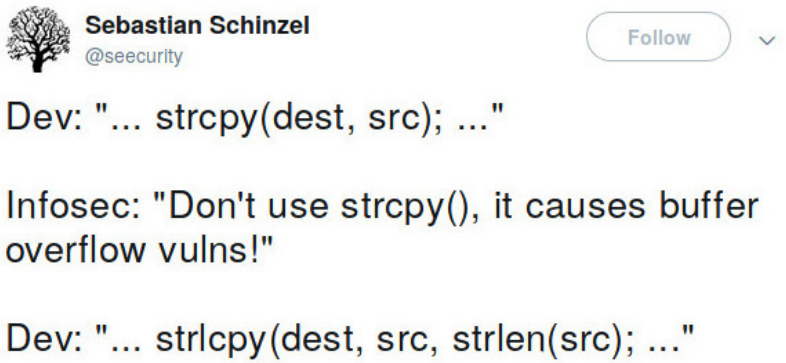


What is checked: For statically sized buffers the compiler can check calls to certain functions.

Enable it with `-DFORTIFY_SOURCE=2` (only with optimization).

Mitigations: Fortify source

Transparently fix *insecure* functions to prevent buffer overflows (memcpy, memset, strcpy, ...).



What is checked: For statically sized buffers the compiler can check calls to certain functions.

Enable it with `-DFORTIFY_SOURCE=2` (only with optimization).

Enabled since SUSE Linux Enterprise Server 10

Mitigations: Fortify source

```c
void fun(char *s) {
  char buf[0x100];
  strcpy(buf, s);
  /* Don't allow gcc to optimise away the buf */
  asm volatile("" :: "m" (buf));
}

int main(int argc, char **argv)
{
  fun( argv[1] );

  return EXIT_SUCCESS;
}
```

Example based on Matthias' work

Mitigations: Fortify source

```
000000000000006b0 <fun>:
 6b0:   55                      push   rbp
 6b1:   48 89 e5                mov    rbp,rsp
 6b4:   48 81 ec 10 01 00 00    sub    rsp,0x110
 6bb:   48 89 bd f8 fe ff ff    mov    QWORD PTR [rbp-0x108],rdi
 6c2:   48 8b 95 f8 fe ff ff    mov    rdx,QWORD PTR [rbp-0x108]
 6c9:   48 8d 85 00 ff ff ff    lea    rax,[rbp-0x100]
 6d0:   48 89 d6                mov    rsi,rdx
 6d3:   48 89 c7                mov    rdi,rax
 6d6:   e8 85 fe ff ff          call   560 <strcpy@plt>
 6db:   90                      nop
 6dc:   c9                      leave
 6dd:   c3                      ret
```

Mitigations: Fortify source

`gcc -o fortify -O2 -D_FORTIFY_SOURCE=2 fortify.c`

```
0000000000000700 <fun>:
 700:   48 81 ec 08 01 00 00    sub    rsp,0x108
 707:   48 89 fe                mov    rsi,rdi
 70a:   ba 00 01 00 00          mov    edx,0x100
 70f:   48 89 e7                mov    rdi,rsp
 712:   e8 69 fe ff ff          call   580 <__strcpy_chk@plt>
 717:   48 81 c4 08 01 00 00    add    rsp,0x108
 71e:   c3                      ret
 71f:   90                      nop
```

Mitigations: Fortify source

```
gcc -o fortify -O2 -D_FORTIFY_SOURCE=2 fortify.c
```

```
0000000000000700 <fun>:
 700:   48 81 ec 08 01 00 00    sub    rsp,0x108
 707:   48 89 fe                mov    rsi,rdi
 70a:   ba 00 01 00 00          mov    edx,0x100
 70f:   48 89 e7                mov    rdi,rsp
 712:   e8 69 fe ff ff          call   580 <__strcpy_chk@plt>
 717:   48 81 c4 08 01 00 00    add    rsp,0x108
 71e:   c3                      ret
 71f:   90                      nop
```

Demo time

Limitation of fortify source

Limitations / problems:

Limitation of fortify source

Limitations / problems:

- Limited to some functions/situations

Limitation of fortify source

Limitations / problems:

- Limited to some functions/situations
- Can still lead to DoS

Limitation of fortify source

Limitations / problems:

- Limited to some functions/situations
- Can still lead to DoS
- Developers might keep using these functions

Limitation of fortify source

Limitations / problems:

- Limited to some functions/situations
- Can still lead to DoS
- Developers might keep using these functions

But it comes with almost no cost, so enable it

Mitigations: ASLR

ASLR: Address space layout randomization

Mitigations: ASLR

ASLR: Address space layout randomization

Memory segments (stack, heap and code) are loaded at random locations

Mitigations: ASLR

ASLR: Address space layout randomization

Memory segments (stack, heap and code) are loaded at random locations

Atttackers don't know return addresses into exploit code or C library code reliably any more

Mitigations: ASLR

```
bash -c 'cat /proc/$$/maps'
56392d605000-56392d60d000 r-xp 00000000 fe:01 12058638  /bin/cat
<snip>
56392dd05000-56392dd26000 rw-p 00000000 00:00 0         [heap]
7fb2bd101000-7fb2bd296000 r-xp 00000000 fe:01 4983399
     /lib/x86_64-linux-gnu/libc-2.24.so
<snip>
7fb2bd6b2000-7fb2bd6b3000 r--p 00000000 fe:01 1836878
     /usr/lib/locale/en_AG/LC_MESSAGES/SYS_LC_MESSAGES
<snip>
7fffd5c36000-7fffd5c57000 rw-p 00000000 00:00 0         [stack]
7fffd5ce9000-7fffd5ceb000 r--p 00000000 00:00 0         [vvar]
7fffd5ceb000-7fffd5ced000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Mitigations: ASLR

```
bash -c 'cat /proc/$$/maps'
56392d605000-56392d60d000 r-xp 00000000 fe:01 12058638   /bin/cat
<snip>
56392dd05000-56392dd26000 rw-p 00000000 00:00 0          [heap]
7fb2bd101000-7fb2bd296000 r-xp 00000000 fe:01 4983399
     /lib/x86_64-linux-gnu/libc-2.24.so
<snip>
7fb2bd6b2000-7fb2bd6b3000 r--p 00000000 fe:01 1836878
     /usr/lib/locale/en_AG/LC_MESSAGES/SYS_LC_MESSAGES
<snip>
7fffd5c36000-7fffd5c57000 rw-p 00000000 00:00 0          [stack]
7fffd5ce9000-7fffd5ceb000 r--p 00000000 00:00 0          [vvar]
7fffd5ceb000-7fffd5ced000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

```
for i in `seq 1 5`; do bash -c 'cat /proc/$$/maps | grep stack'; done
7ffcb8e0f000-7ffcb8e30000 rw-p 00000000 00:00 0 [stack]
7fff64dc9000-7fff64dea000 rw-p 00000000 00:00 0 [stack]
7ffc3b408000-7ffc3b429000 rw-p 00000000 00:00 0 [stack]
7ffcee799000-7ffcee7ba000 rw-p 00000000 00:00 0 [stack]
7ffd4b904000-7ffd4b925000 rw-p 00000000 00:00 0 [stack]
```

Mitigations: ASLR

`cat /proc/sys/kernel/randomize_va_space` shows you the current settings for your system.

- **0**: No randomization
- **1**: Randomize positions of the stack, VDSO page, and shared memory regions
- **2**: Randomize positions of the stack, VDSO page, shared memory regions, and the data segment

Mitigations: ASLR

`cat /proc/sys/kernel/randomize_va_space` shows you the current settings for your system.

- **0**: No randomization
- **1**: Randomize positions of the stack, VDSO page, and shared memory regions
- **2**: Randomize positions of the stack, VDSO page, shared memory regions, and the data segment

To get the full benefit you need to compile your binaries with `-fPIE`

Mitigations: ASLR

Limitations:

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines
- Limited entropy on 32 bit systems

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines
- Limited entropy on 32 bit systems
- Brute forcing still an issue if restart is not handled properly.

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines
- Limited entropy on 32 bit systems
- Brute forcing still an issue if restart is not handled properly.
- Can be circumvented by chaining an information leak into the exploit

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines
- Limited entropy on 32 bit systems
- Brute forcing still an issue if restart is not handled properly.
- Can be circumvented by chaining an information leak into the exploit
- Some exotic software might rely on fixed addresses (think inline assembly)

Mitigations: ASLR

Limitations:

- 5 - 10% performance loss on i386 machines
- Limited entropy on 32 bit systems
- Brute forcing still an issue if restart is not handled properly.
- Can be circumvented by chaining an information leak into the exploit
- Some exotic software might rely on fixed addresses (think inline assembly)
- Sometimes you have usable memory locations in registers

Mitigations: No-execute memory

Modern processors support memory to be mapped as non-executable

Mitigations: No-execute memory

Modern processors support memory to be mapped as non-executable

Another term for this feature is NX or W^X

Mitigations: No-execute memory

Modern processors support memory to be mapped as non-executable

Another term for this feature is NX or W^X

The most interesting memory regions for this feature to use are the stack and heap memory regions

Mitigations: No-execute memory

Modern processors support memory to be mapped as non-executable

Another term for this feature is NX or W^X

The most interesting memory regions for this feature to use are the stack and heap memory regions

A stack overflow could still take place, but it is not be possible to *directly* return to a stack address for execution

Mitigations: No-execute memory

Modern processors support memory to be mapped as non-executable

Another term for this feature is NX or W^X

The most interesting memory regions for this feature to use are the stack and heap memory regions

A stack overflow could still take place, but it is not be possible to *directly* return to a stack address for execution

```
bash -c 'cat /proc/$$/maps | grep stack'
7ffcb8e0f000-7ffcb8e30000 rw-p 00000000 00:00 0 [stack]
```

Mitigations: NX

Limitations

Mitigations: NX

Limitations

- Use existing code in the exploited program

Mitigations: NX

Limitations

- Use existing code in the exploited program
- Return to libc: Use existing functions

Mitigations: NX

Limitations

- Use existing code in the exploited program
- Return to libc: Use existing functions
- ROP (Return Oriented Programming): Structure the data on the stack so that instruction sequences ending in `ret` can be used

Mitigations: Stack clash protection

Heap and stack live both in the address space of the process and they grow dynamically

```
bash -c 'cat /proc/$$/maps | egrep "(heap|stack)"'
55dc4ffbe000-55dc4ffdf000 rw-p 00000000 00:00 0                          [heap]
7ffce8c2b000-7ffce8c4c000 rw-p 00000000 00:00 0                          [stack]
```

Mitigations: Stack clash protection

Heap and stack live both in the address space of the process and they grow dynamically

```
bash -c 'cat /proc/$$/maps | egrep "(heap|stack)"'
55dc4ffbe000-55dc4ffdf000 rw-p 00000000 00:00 0                          [heap]
7ffce8c2b000-7ffce8c4c000 rw-p 00000000 00:00 0                          [stack]
```

So what happens if those two meet?

Mitigations: Stack clash protection

Heap and stack live both in the address space of the process and they grow dynamically

```
bash -c 'cat /proc/$$/maps | egrep "(heap|stack)"'
55dc4ffbe000-55dc4ffdf000 rw-p 00000000 00:00 0                          [heap]
7ffce8c2b000-7ffce8c4c000 rw-p 00000000 00:00 0                          [stack]
```

So what happens if those two meet?

Guard page is inserted between stack and heap, causes an segmentation fault if they clash

Mitigations: Stack clash protection

But what if we can skip the guard page?

Mitigations: Stack clash protection

But what if we can skip the guard page?

Bring stack and heap close, then use an allocation $>$ one page to jump the guard page

Mitigations: Stack clash protection

But what if we can skip the guard page?

Bring stack and heap close, then use an allocation $>$ one page to jump the guard page

After that you can write to the stack to modify data on the heap or the other way around

Mitigations: Stack clash protection

To prevent this compile code with: `-fstack-clash-protection`

Mitigations: Stack clash protection

To prevent this compile code with: `-fstack-clash-protection`

Ensures access to every page when doing large allocations

Mitigations: Stack clash protection

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

void
fancyprint( char *str1, char *str2 )
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  stpcpy (stpcpy (name, str1), str2);
  printf("%s\n", name );
}

int
main( int argc, char *argv[]) {
  if ( argc < 3 ) {
    return EXIT_FAILURE;
  }

  fancyprint( argv[1], argv[2] );
  return EXIT_SUCCESS;
}
```

Mitigations: Stack clash protection

```
<+39>:    call   0x400500 <strlen@plt>
<+44>:    add    rax,rbx
<+47>:    add    rax,0x1
<+51>:    lea    rdx,[rax+0xf]
<+55>:    mov    eax,0x10
<+60>:    sub    rax,0x1
<+64>:    add    rax,rdx
<+67>:    mov    ecx,0x10
<+72>:    mov    edx,0x0
<+77>:    div    rcx
<+80>:    imul   rax,rax,0x10
<+84>:    sub    rsp,rax
<+87>:    mov    rax,rsp
<+90>:    add    rax,0xf
<+94>:    shr    rax,0x4
<+98>:    shl    rax,0x4
```

Mitigations: Stack clash protection

```
<+39>:    call   0x400500 <strlen@plt>
<+44>:    add    rax,rbx
<+47>:    add    rax,0x1
<+51>:    lea    rdx,[rax+0xf]
<+55>:    mov    eax,0x10
<+60>:    sub    rax,0x1
<+64>:    add    rax,rdx
<+67>:    mov    ecx,0x10
<+72>:    mov    edx,0x0
<+77>:    div    rcx
<+80>:    imul   rax,rax,0x10
<+84>:    sub    rsp,rax
<+87>:    mov    rax,rsp
<+90>:    add    rax,0xf
<+94>:    shr    rax,0x4
<+98>:    shl    rax,0x4
```

Mitigations: Stack clash protection

```
<+39>:    call   0x400500 <strlen@plt>
<+44>:    add    rax,rbx
<+47>:    add    rax,0x1
<+51>:    lea    rdx,[rax+0xf]
<+55>:    mov    eax,0x10
<+60>:    sub    rax,0x1
<+64>:    add    rax,rdx
<+67>:    mov    ecx,0x10
<+72>:    mov    edx,0x0
<+77>:    div    rcx
<+80>:    imul   rax,rax,0x10
<+84>:    sub    rsp,rax
<+87>:    mov    rax,rsp
<+90>:    add    rax,0xf
<+94>:    shr    rax,0x4
<+98>:    shl    rax,0x4
```

Mitigations: Stack clash protection

```
40060e:       e8 ed fe ff ff          call   400500 <strlen@plt>
400613:       48 01 d8                add    rax,rbx
400616:       48 83 c0 01             add    rax,0x1
40061a:       48 8d 50 0f             lea    rdx,[rax+0xf]
40061e:       b8 10 00 00 00          mov    eax,0x10
400623:       48 83 e8 01             sub    rax,0x1
400627:       48 01 d0                add    rax,rdx
40062a:       b9 10 00 00 00          mov    ecx,0x10
40062f:       ba 00 00 00 00          mov    edx,0x0
400634:       48 f7 f1                div    rcx
400637:       48 6b c0 10             imul   rax,rax,0x10
40063b:       48 89 e2                mov    rdx,rsp
40063e:       48 85 c0                test   rax,rax
400641:       74 28                   je     40066b <fancyprint+0x84>
400643:       48 3d 00 10 00 00       cmp    rax,0x1000
400649:       72 16                   jb     400661 <fancyprint+0x7a>
40064b:       48 2d 00 10 00 00       sub    rax,0x1000
400651:       48 81 ec 00 10 00 00    sub    rsp,0x1000
400658:       48 89 e2                mov    rdx,rsp
40065b:       48 83 0a 00             or     QWORD PTR [rdx],0x0
40065f:       eb e2                   jmp    400643 <fancyprint+0x5c>
400661:       48 29 c4                sub    rsp,rax
```

Mitigations: Stack clash protection

```
40060e:       e8 ed fe ff ff          call   400500 <strlen@plt>
400613:       48 01 d8                add    rax,rbx
400616:       48 83 c0 01             add    rax,0x1
40061a:       48 8d 50 0f             lea    rdx,[rax+0xf]
40061e:       b8 10 00 00 00          mov    eax,0x10
400623:       48 83 e8 01             sub    rax,0x1
400627:       48 01 d0                add    rax,rdx
40062a:       b9 10 00 00 00          mov    ecx,0x10
40062f:       ba 00 00 00 00          mov    edx,0x0
400634:       48 f7 f1                div    rcx
400637:       48 6b c0 10             imul   rax,rax,0x10
40063b:       48 89 e2                mov    rdx,rsp
40063e:       48 85 c0                test   rax,rax
400641:       74 28                   je     40066b <fancyprint+0x84>
400643:       48 3d 00 10 00 00       cmp    rax,0x1000
400649:       72 16                   jb     400661 <fancyprint+0x7a>
40064b:       48 2d 00 10 00 00       sub    rax,0x1000
400651:       48 81 ec 00 10 00 00    sub    rsp,0x1000
400658:       48 89 e2                mov    rdx,rsp
40065b:       48 83 0a 00             or     QWORD PTR [rdx],0x0
40065f:       eb e2                   jmp    400643 <fancyprint+0x5c>
400661:       48 29 c4                sub    rsp,rax
```

Mitigations: Stack clash protection

```
40060e:       e8 ed fe ff ff          call   400500 <strlen@plt>
400613:       48 01 d8                add    rax,rbx
400616:       48 83 c0 01             add    rax,0x1
40061a:       48 8d 50 0f             lea    rdx,[rax+0xf]
40061e:       b8 10 00 00 00          mov    eax,0x10
400623:       48 83 e8 01             sub    rax,0x1
400627:       48 01 d0                add    rax,rdx
40062a:       b9 10 00 00 00          mov    ecx,0x10
40062f:       ba 00 00 00 00          mov    edx,0x0
400634:       48 f7 f1                div    rcx
400637:       48 6b c0 10             imul   rax,rax,0x10
40063b:       48 89 e2                mov    rdx,rsp
40063e:       48 85 c0                test   rax,rax
400641:       74 28                   je     40066b <fancyprint+0x84>
400643:       48 3d 00 10 00 00       cmp    rax,0x1000
400649:       72 16                   jb     400661 <fancyprint+0x7a>
40064b:       48 2d 00 10 00 00       sub    rax,0x1000
400651:       48 81 ec 00 10 00 00    sub    rsp,0x1000
400658:       48 89 e2                mov    rdx,rsp
40065b:       48 83 0a 00             or     QWORD PTR [rdx],0x0
40065f:       eb e2                   jmp    400643 <fancyprint+0x5c>
400661:       48 29 c4                sub    rsp,rax
```

Mitigations: Stack clash protection

```
40060e:       e8 ed fe ff ff          call   400500 <strlen@plt>
400613:       48 01 d8                add    rax,rbx
400616:       48 83 c0 01             add    rax,0x1
40061a:       48 8d 50 0f             lea    rdx,[rax+0xf]
40061e:       b8 10 00 00 00          mov    eax,0x10
400623:       48 83 e8 01             sub    rax,0x1
400627:       48 01 d0                add    rax,rdx
40062a:       b9 10 00 00 00          mov    ecx,0x10
40062f:       ba 00 00 00 00          mov    edx,0x0
400634:       48 f7 f1                div    rcx
400637:       48 6b c0 10             imul   rax,rax,0x10
40063b:       48 89 e2                mov    rdx,rsp
40063e:       48 85 c0                test   rax,rax
400641:       74 28                   je     40066b <fancyprint+0x84>
400643:       48 3d 00 10 00 00       cmp    rax,0x1000
400649:       72 16                   jb     400661 <fancyprint+0x7a>
40064b:       48 2d 00 10 00 00       sub    rax,0x1000
400651:       48 81 ec 00 10 00 00    sub    rsp,0x1000
400658:       48 89 e2                mov    rdx,rsp
40065b:       48 83 0a 00             or     QWORD PTR [rdx],0x0
40065f:       eb e2                   jmp    400643 <fancyprint+0x5c>
400661:       48 29 c4                sub    rsp,rax
```

Limitations of stack clash protection

Limitations:

Limitations of stack clash protection

Limitations:

- Only works for calculated allocations

Limitations of stack clash protection

Limitations:

- Only works for calculated allocations
- Some edge cases (think inline assembly) not covered

Limitations of stack clash protection

Limitations:

- Only works for calculated allocations
- Some edge cases (think inline assembly) not covered
- Minor performance loss

Limitations of stack clash protection

Limitations:

- Only works for calculated allocations
- Some edge cases (think inline assembly) not covered
- Minor performance loss

High value mitigation with almost no downsides. See CVE-2018-16864 ("System down")

Limitations of stack clash protection

Limitations:

- Only works for calculated allocations
- Some edge cases (think inline assembly) not covered
- Minor performance loss

High value mitigation with almost no downsides. See CVE-2018-16864 ("System down")

Enabled since SUSE Linux Enterprise Server 15, all SLE 12 updates receive the protection automatically

RELRO

RELRO: RELocation Read Only

RELRO

RELRO: RELocation Read Only

Relocations is a level of indirection to allow code to run at arbitrary locations

RELRO

RELRO: RELocation Read Only

Relocations is a level of indirection to allow code to run at arbitrary locations

If binaries are

- are compiled with `-fPIE`, `-fPIC`
- use dynamic libraries

relocations are necessary

RELRO

Example code:

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

int
main( int argc, char *argv[] ) {
  printf("Hello world\n");

  printf("Nice to see you again\n");
}
```

RELRO

Example code:

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

int
main( int argc, char *argv[] ) {
  printf("Hello world\n");

  printf("Nice to see you again\n");
}
```

First `printf` will need to go through relocation process, second `printf` can jump directly.

Demo time

RELRO

The entries in these tables can be overwritten to execute different code

RELRO

The entries in these tables can be overwritten to execute different code

Partial RELRO (protects against overflows in global variables) since SUSE Linux Enterprise Server 10

RELRO

The entries in these tables can be overwritten to execute different code

Partial RELRO (protects against overflows in global variables) since SUSE Linux Enterprise Server 10

Full RELRO is in testing in Factory

RELRO

The entries in these tables can be overwritten to execute different code

Partial RELRO (protects against overflows in global variables) since SUSE Linux Enterprise Server 10

Full RELRO is in testing in Factory

Drawbacks of full RELRO:

- Startup time is increased in large programs (thousands of symbols)

Outlook

Exploitation got harder, but this is an ongoing struggle

Outlook

Exploitation got harder, but this is an ongoing struggle

ROP is used in a lot of modern exploits:

- Shadow stacks
- (Hardware) control flow integrity (CFI)

Outlook

Exploitation got harder, but this is an ongoing struggle

ROP is used in a lot of modern exploits:

- Shadow stacks
- (Hardware) control flow integrity (CFI)

These mitigations are currently rather costly, hard to convince users to take the hit

Outlook

Exploitation got harder, but this is an ongoing struggle

ROP is used in a lot of modern exploits:

- Shadow stacks
- (Hardware) control flow integrity (CFI)

These mitigations are currently rather costly, hard to convince users to take the hit

So keep your systems updated

Thank you for your attention!

Questions?

Slides at https://users.suse.com/~jsegitz/talks/2019.04-susecon.pdf
Or just scan: