SUSEcon digital '20

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

Johannes Segitz, SUSE security team

SUSE

Who am I?

Johannes Segitz, security engineer (Nuremberg, Germany)

- Code review
- Product pentesting

Outline

Buffer overflows and protections:

- Stack canaries
- Fortify source
- Address space layout randomization
- No-execute memory (NX, W^X)
- Stack clash protection without example
- RELRO

Outline

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

Outline

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

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

Outline

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

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:

```
1 #include <string.h>
2
3 int main(int argc, char **argv) {
4   char buffer[20];
5
6   strcpy(buffer, argv[1]);
7
8   return EXIT_SUCCESS;
9 }
```

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

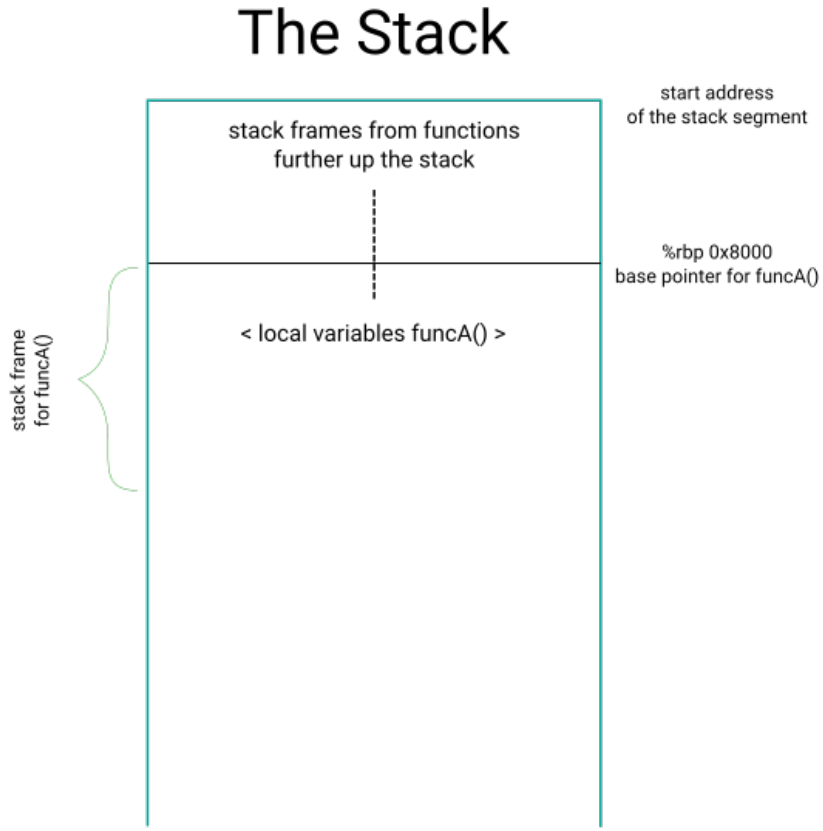
The problem is that for a given buffer size too much data is placed in there

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?

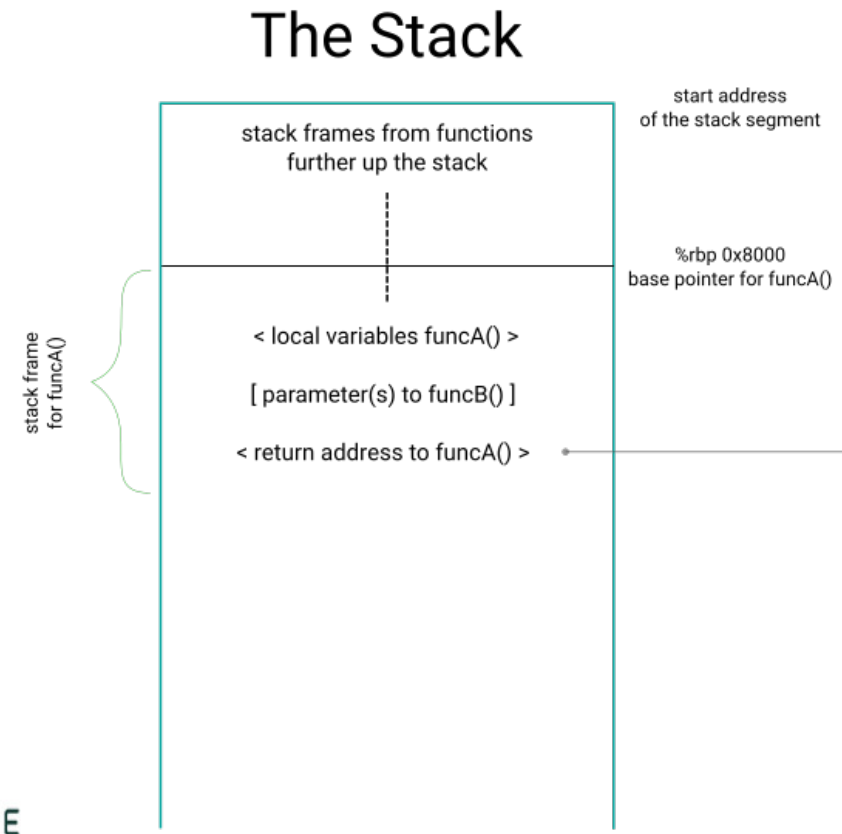
Because in data of the application and control information about execution is mixed




```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?

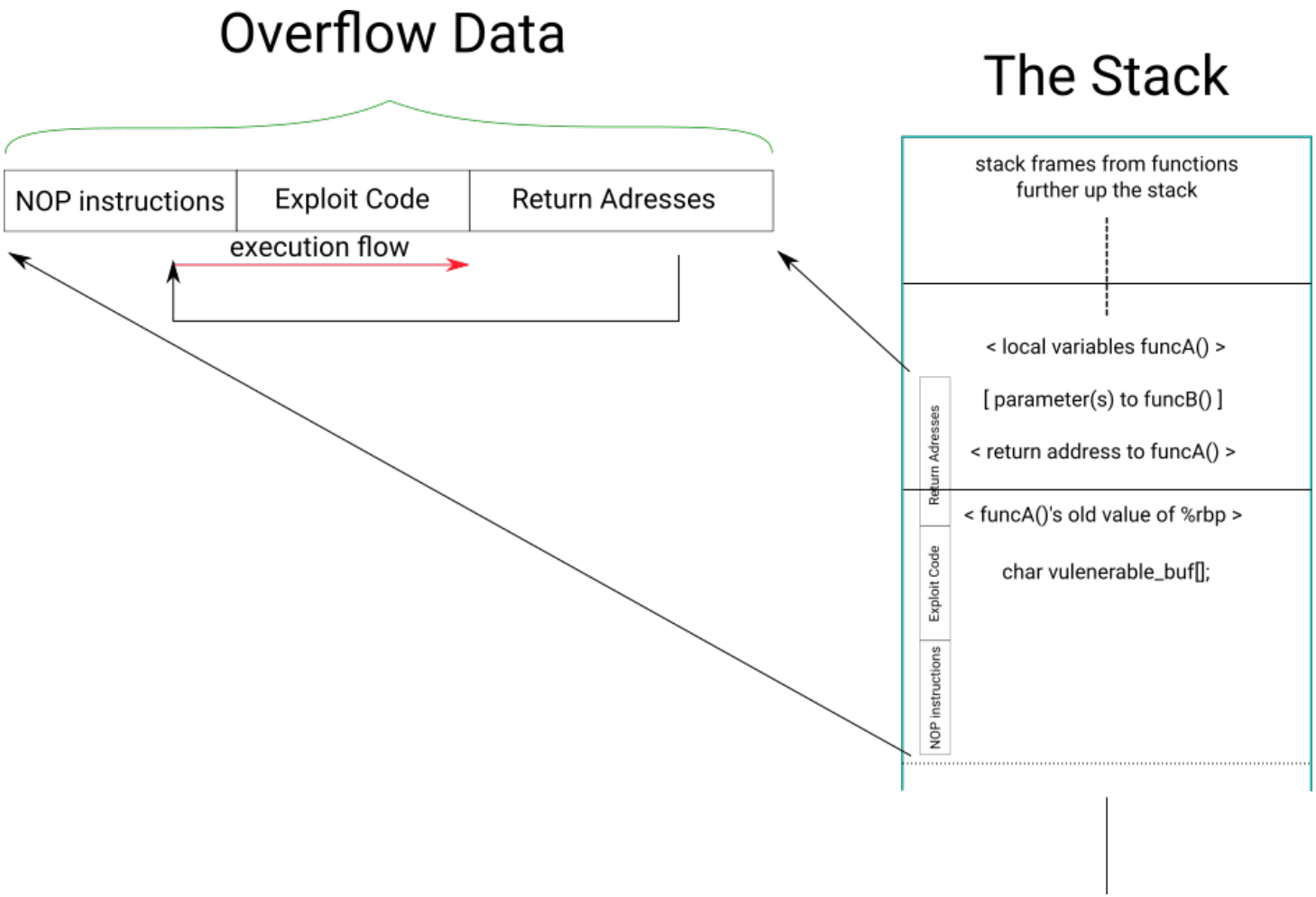
If a buffer overflow happens this control information can be overwritten

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 ;)

32 bit exploitation

```
1  #include <unistd.h>
2
3  void vulnerable( void ) {
4    char buffer[256];
5
6    read(0, buffer, 512);
7
8    return;
9  }
10
11 int main(int argc, char **argv) {
12   vulnerable();
13
14   return EXIT_SUCCESS;
15 }
```

32 bit exploitation

Demo time

Stack canaries

Stack canaries

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

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

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

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

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

Stack canaries

Four variants in gcc:

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

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

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

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`

Stack canaries

Short reminder of the example code:

```
1  #include <string.h>
2
3  int main(int argc, char **argv)
4  {
5    char buffer[20];
6
7    strcpy(buffer, argv[1]);
8
9    return EXIT_SUCCESS;
10 }
```

Stack canaries

```
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
```

Stack canaries

```
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
```

Stack canaries

```
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
```

Stack canaries

```
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
```

Stack canaries

Demo time

Stack canaries

Limitations:

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

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

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

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

Stack canaries

Limitations:

- No protection for inlined functions

Stack canaries

Limitations:

- No protection for inlined functions
- Can be used to cause DoS

ASLR

ASLR: Address space layout randomization

ASLR

ASLR: Address space layout randomization

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

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

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]
```

ASLR

```
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]
```

ASLR

Limitations:

- 5 - 10% performance loss on i386 machines

ASLR

Limitations:

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

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.

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

ASLR

Limitations:

- Some exotic software might rely on fixed addresses (think inline assembly)

ASLR

Limitations:

- Some exotic software might rely on fixed addresses (think inline assembly)
- Sometimes you have usable memory locations in registers

No-execute memory

Modern processors support memory to be mapped as non-executable

No-execute memory

Modern processors support memory to be mapped as non-executable

Another term for this feature is NX or W^X

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

No-execute memory

No-execute memory

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

No-execute memory

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]
```

NX

Limitations

- Use existing code in the exploited program

NX

Limitations

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

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

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]
```

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?

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

Stack clash protection

But what if we can skip the guard page?

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

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

Stack clash protection

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

Stack clash protection

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

Ensures access to every page when doing large allocations

Stack clash protection

Limitations:

- Only works for calculated allocations

Stack clash protection

Limitations:

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

Stack clash protection

Limitations:

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

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")

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)

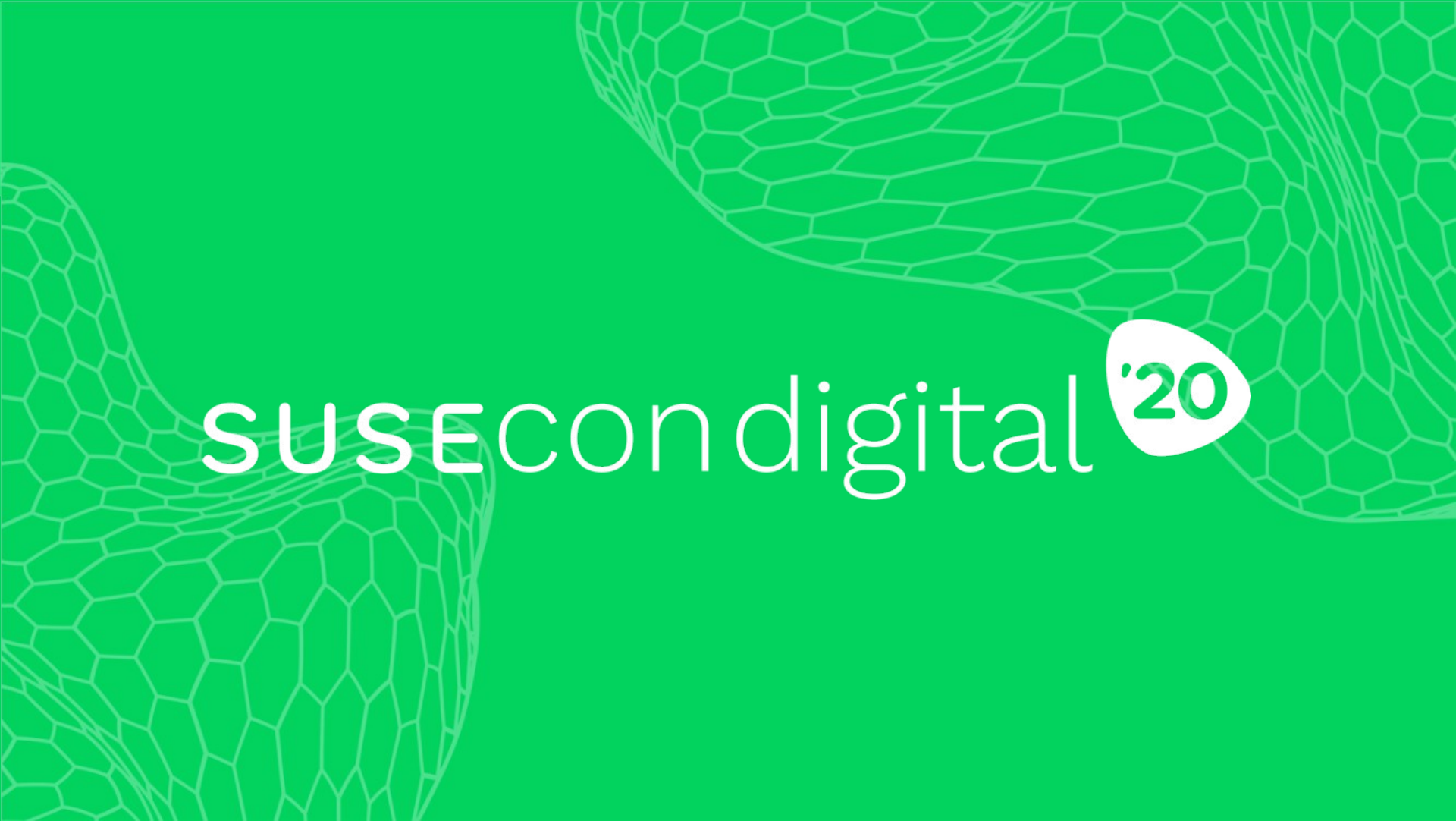
These mitigations are currently rather costly, hard to convince users to take the hit

So keep your systems updated

Q&A

General Disclaimer

This document is not to be construed as a promise by any participating company to develop, deliver, or market a product. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. SUSE makes no representations or warranties with respect to the contents of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. The development, release, and timing of features or functionality described for SUSE products remains at the sole discretion of SUSE. Further, SUSE reserves the right to revise this document and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. All SUSE marks referenced in this presentation are trademarks or registered trademarks of SUSE, LLC, Inc. in the United States and other countries. All third-party trademarks are the property of their respective owners.


SUSEcon digital
'20