

Sécurité des logiciels

Compilation hardening

Samuel Thibault <samuel.thibault@u-bordeaux.fr>

CC-BY-NC-SA

GOT overwriting

Calling a library function?

```
#include <stdio.h>
int main(void) {
    puts("foo");
    return 0;
}
```

How does this actually work?

Calling a library function?

```
...  
extern int puts(const char *s);  
...  
int main(void) {  
    puts("foo");  
    return 0;  
}
```

Calling a library function?

```
...  
extern int puts(const char *s);
```

```
...  
int main(void) {  
    puts("foo");  
    return 0;  
}
```

```
gcc test.c -o test.o -c ; objdump -d test.o  
[...]  
1040:    e8 00 00 00 00 callq 10 <main+0x10>
```

I.e. leaves a “hole”, a “relocation”: no idea what it should be yet

Calling a library function?

```
...  
extern int puts(const char *s);
```

```
...  
int main(void) {  
    puts("foo");  
    return 0;  
}
```

```
gcc test.o -o test ; objdump -d test
```

```
[...]
```

```
1040:    e8 eb fe ff ff callq 1030 <puts@plt>
```

“Filled” the hole

The thing is: we don't know where libc will be in memory!

Calling a library function?

<main>

```
1040: e8 eb fe ff ff      callq 1030 <puts@plt>
```

Disassembly of section .plt :

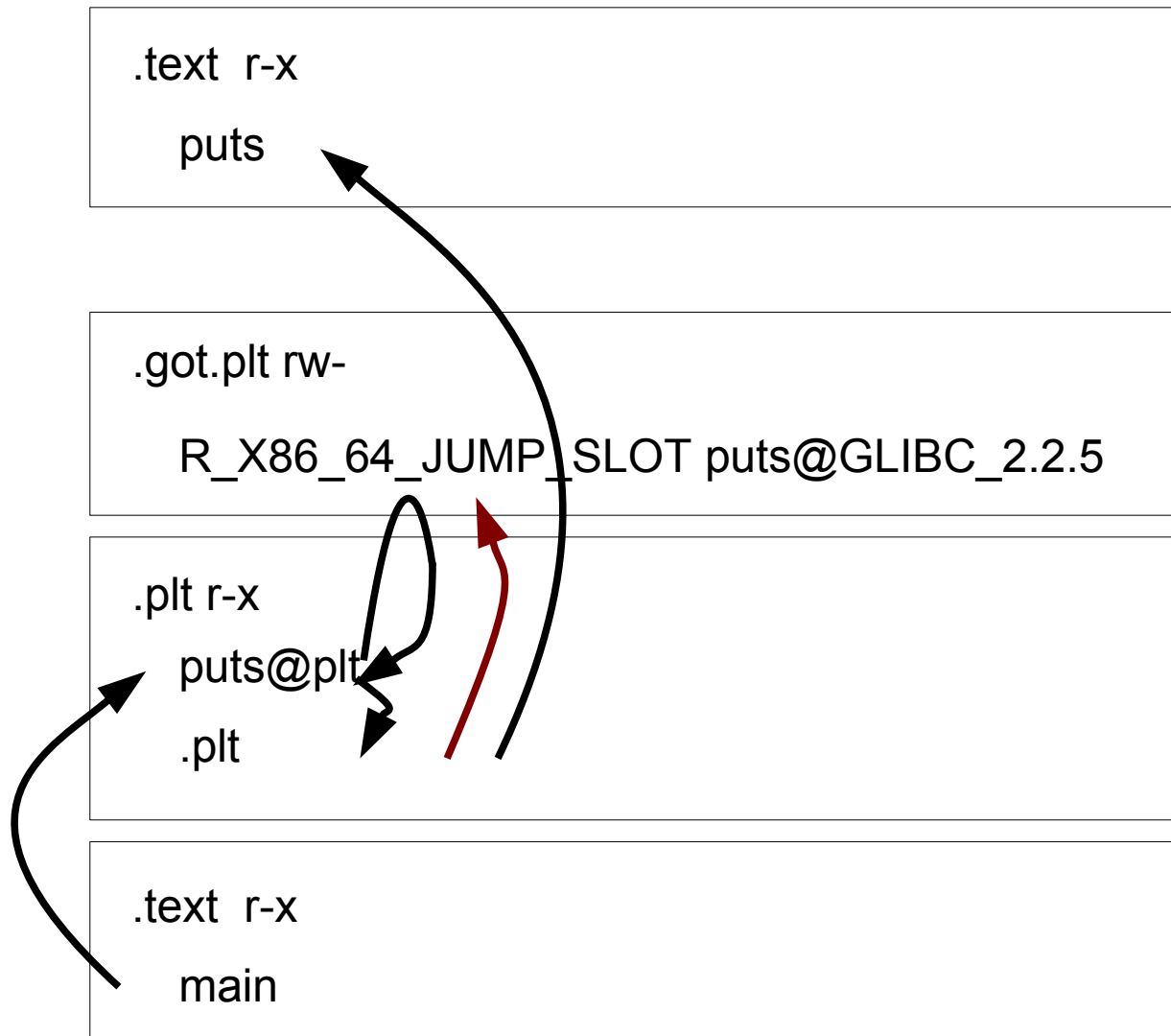
<puts@plt>:

```
1030: ff 25 e2 2f 00 00    jmpq    *0x2fe2(%rip)
                                     # 4018 <puts@GLIBC_2.2.5>
1036: 68 00 00 00 00      pushq   $0x0
103b: e9 e0 ff ff ff      jmpq    1020 <.plt>
```

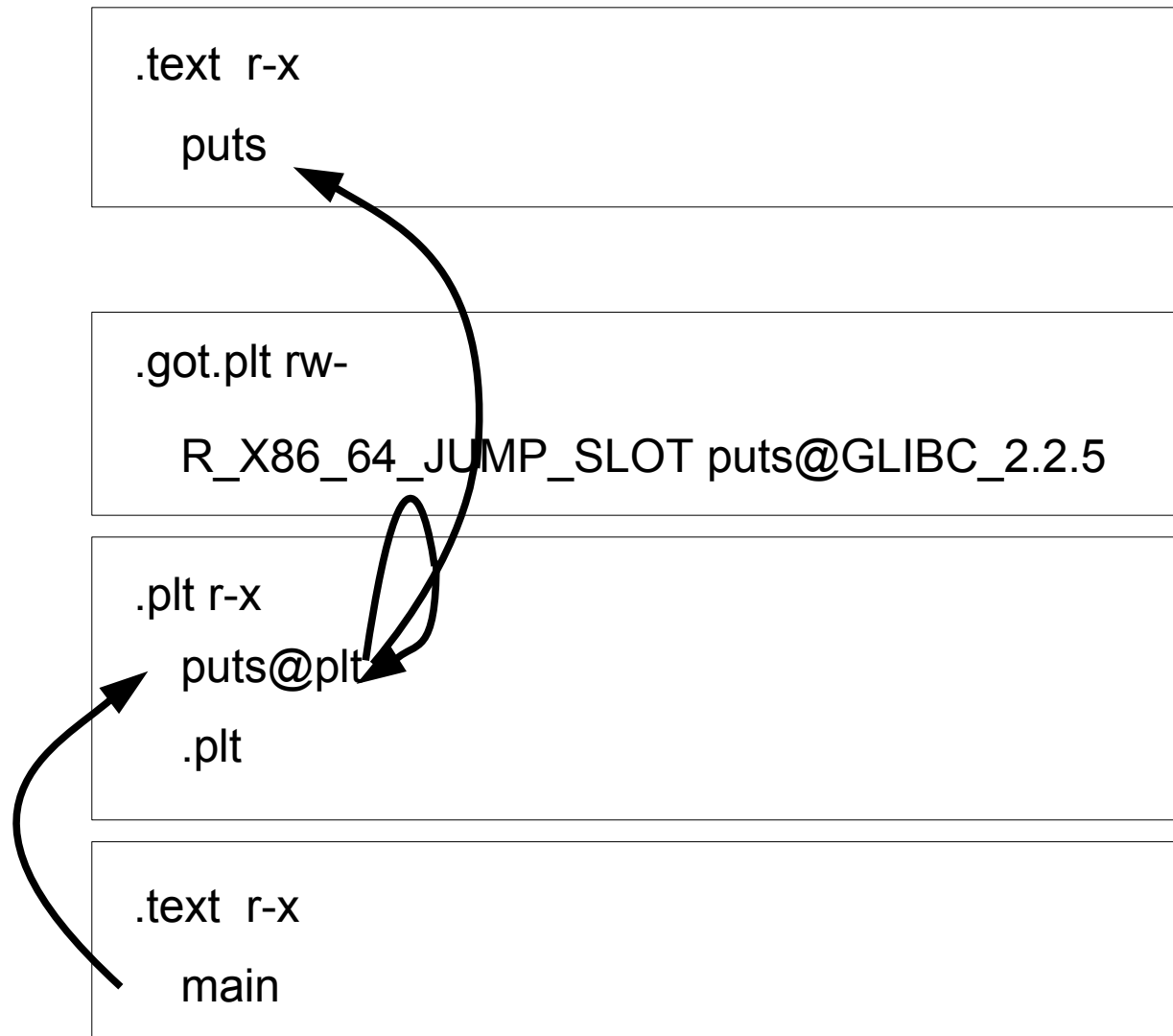
Disassembly of section .got.plt :

```
4018: 36 10 00 00 00 00
```

Calling a library function?



Calling a library function?



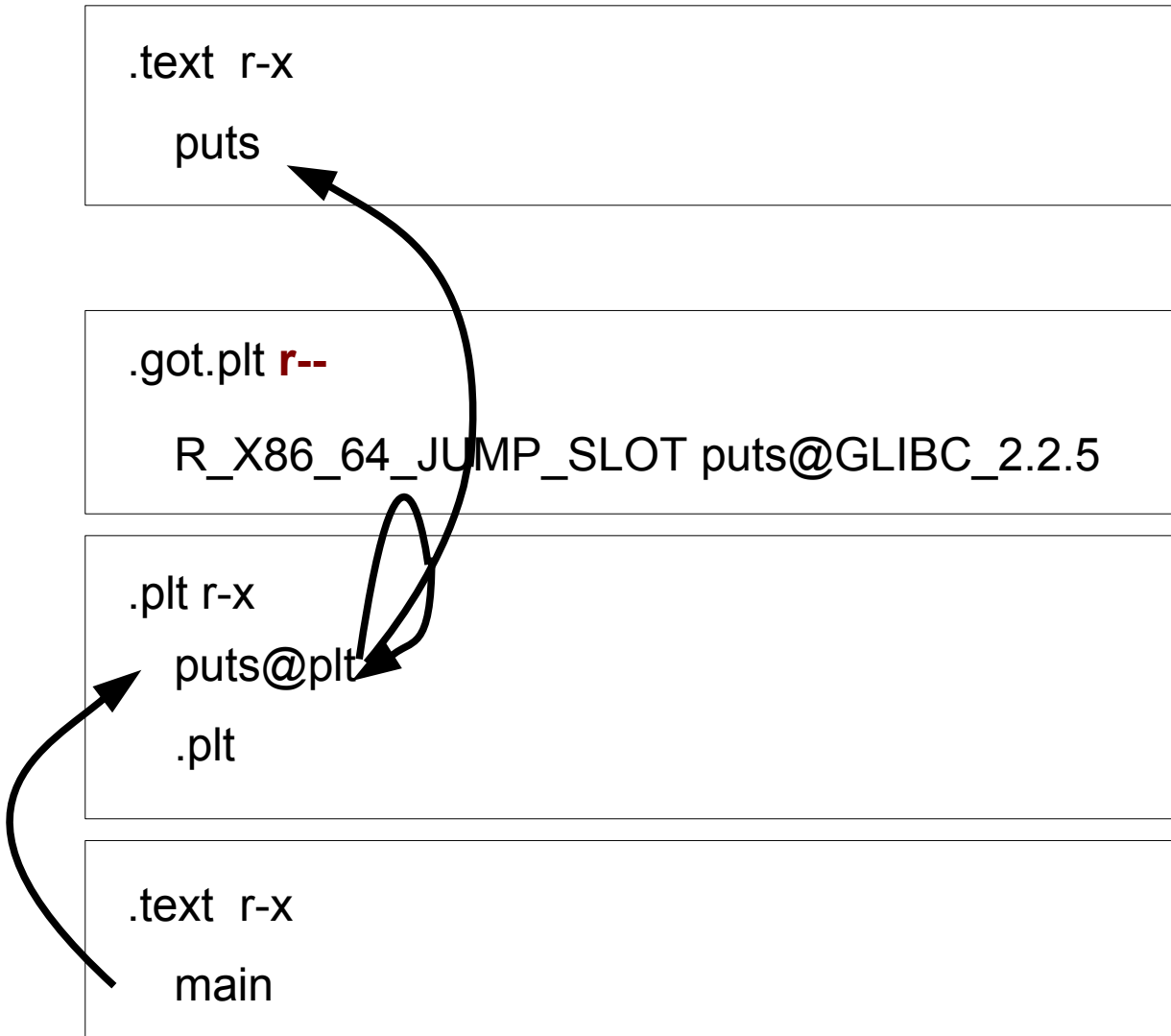
Calling a library function?

```
.text r-x  
puts
```

```
.got.plt r--  
R_X86_64_JUMP_SLOT puts@GLIBC_2.2.5
```

```
.plt r-x  
puts@plt  
.plt
```

```
.text r-x  
main
```



Calling a library function?

- Library calls through GOT
- By default, function names resolved lazily
- By default, GOT writable
 - attack target
 - making it read-only after program load
- `-Wl,-z,relro` `-Wl,-z,now`
- One example of build-time fortification option
- Possibly use prelink



Fortifying the build

Fortifying the build

- Checks in the generated code
- Checks in libc
- Guard areas
- Strengthened linking
- Strengthened memory layout

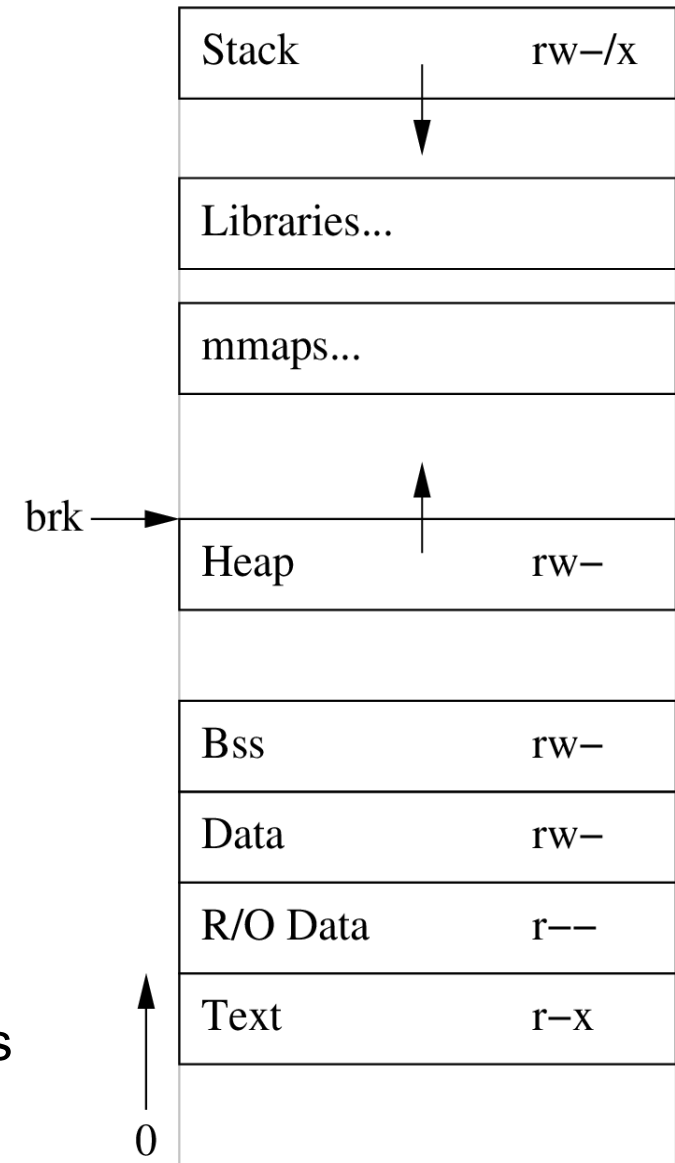
- More or less execution-time cost
- But considered worth the price



Fortifying the build Memory layout

Memory layout

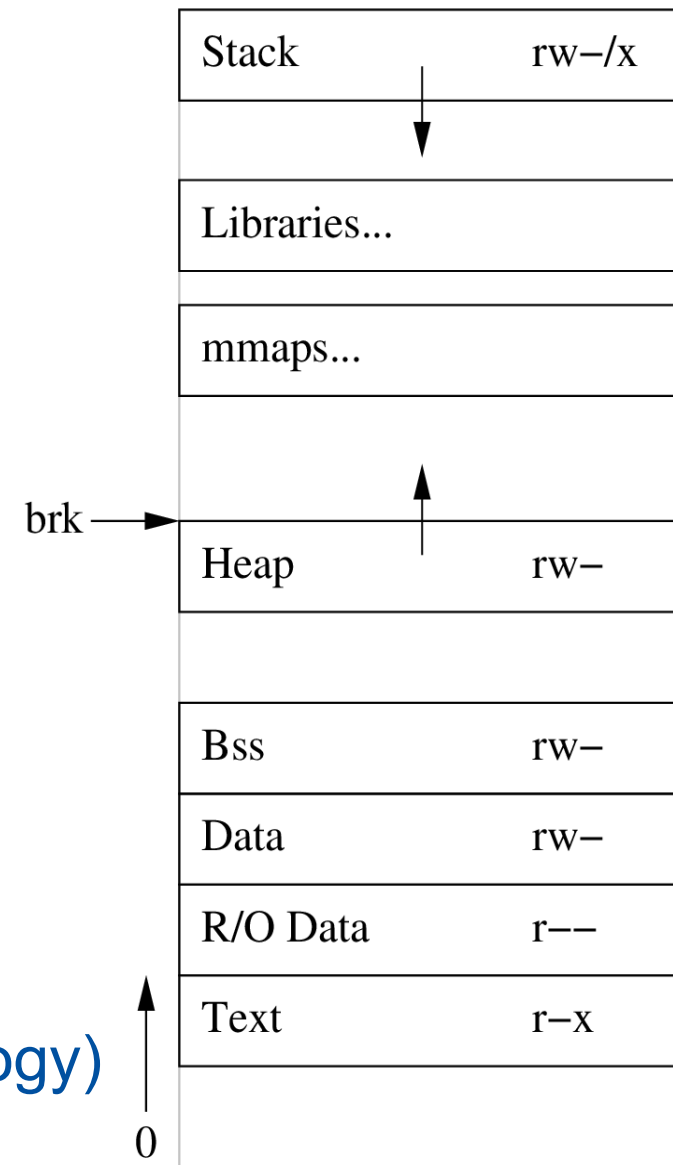
- **ASLR!**
 - Nowadays, all areas at random locations
 - Large holes between regions
 - Requires PIE options for text
 - fPIE -pie**
 - (Already by default nowadays)
- **Non-executable stack**
 - Wl,-z,noexecstack**
 - (Already by default nowadays)
- **Control-related data in read-only regions**
 - Wl,-z,relro -Wl,-z,now**
 - Also use **const** as much as possible
 - e.g. for structures containing methods pointers



Memory layout

Structures containing methods pointers

- Common in large C programs
 - Use const!
- Quite common in C++ programs
 - Object virtual methods!
 - Attack targets
 - **fvtable-verify=std**
 - Verifies the target makes sense for the object
- More generally, all indirect calls / jumps
 - **fcf-protect=branch**
 - Verifies the target makes sense
- Hardware support being added by Intel:
CET (Control-Flow Enforcement Technology)
 - **All** branch targets marked with ENDBR

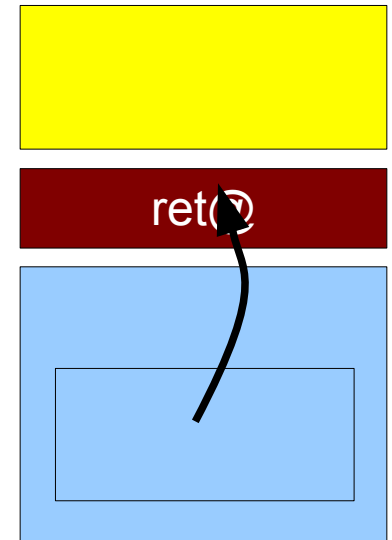




Fortifying the build Stack protection

Stack protection

Stack-based buffer overflow

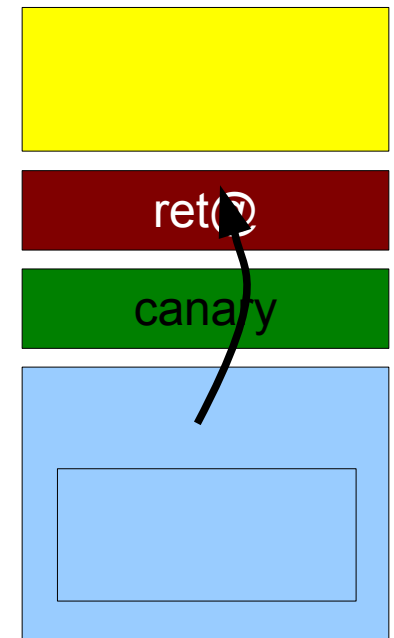


Stack protection

Stack-based buffer overflow

Adding a canary

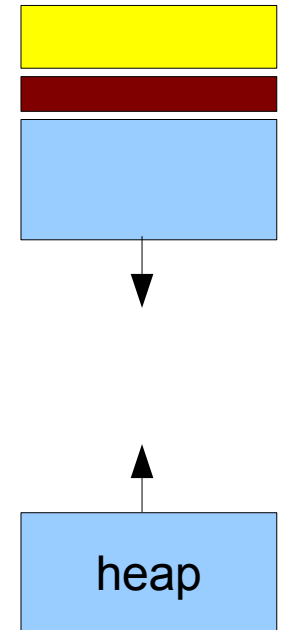
- Overwritten as well
- Checked before `ret` instruction
- Ideally random (and per-thread), or
- **0x000d0aff**
 - Contains a `'\0'`
 - Contains a `'\r' + '\n'`
 - Contains `~'\0'`



Stack protection

Stack clashing

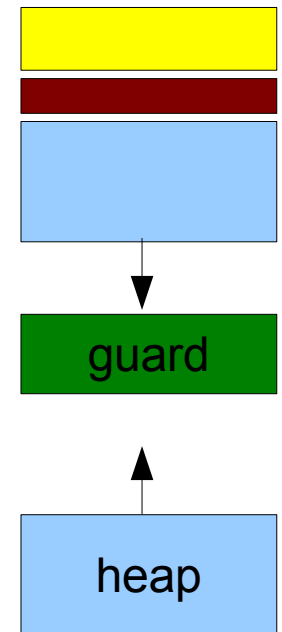
- What happens if stack grows too much?



Stack protection

Stack clashing

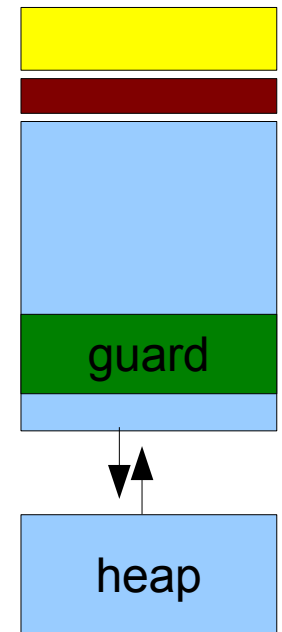
- What happens if stack grows too much?
- Putting a stack guard to prevent clash



Stack protection

Stack clashing

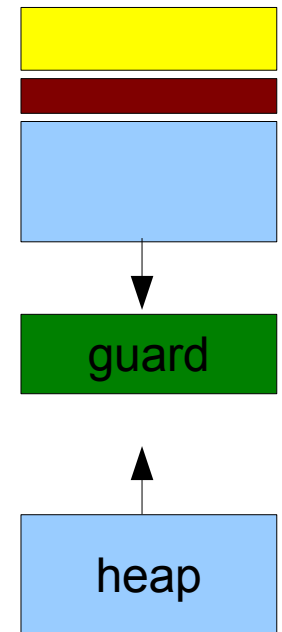
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it



Stack protection

Stack clashing

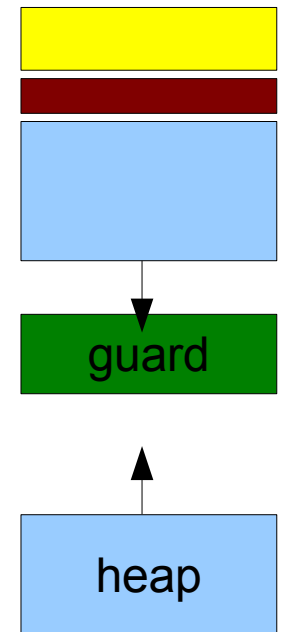
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Stack clashing

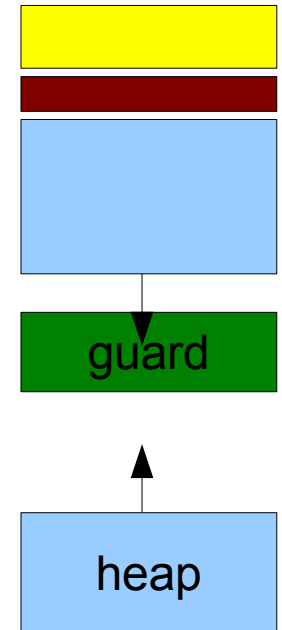
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Stack clashing

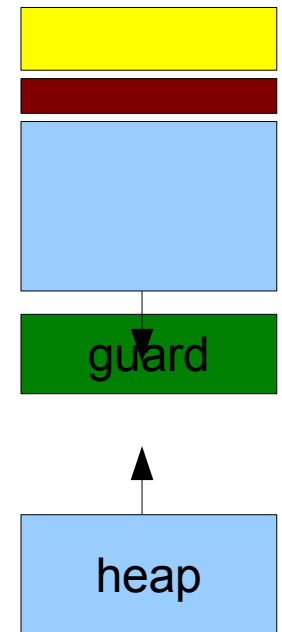
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Stack clashing

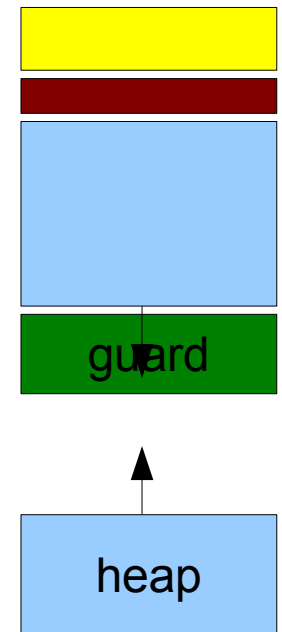
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Stack clashing

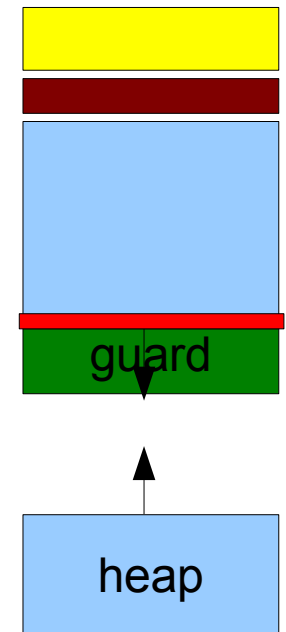
- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Stack clashing

- What happens if stack grows too much?
- Putting a stack guard to prevent clash
- But if really large allocation, could step over it
- Make the compiler allocate precautiously
 - `-fstack-clash-protection`



Stack protection

Control-Flow Protection

- Basic issue: we are not returning where we are supposed to

Also remember Return-Oriented Programming

- Picking up gadgets from libc

```
addl $12,%esp  
ret
```

Basically, erratic control flow

Stack protection

Control-Flow Protection



Shadow stack

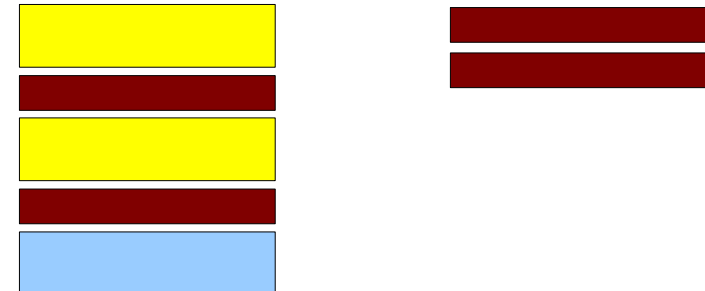
- Replicates return addresses
- Somewhere else in address space

Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space

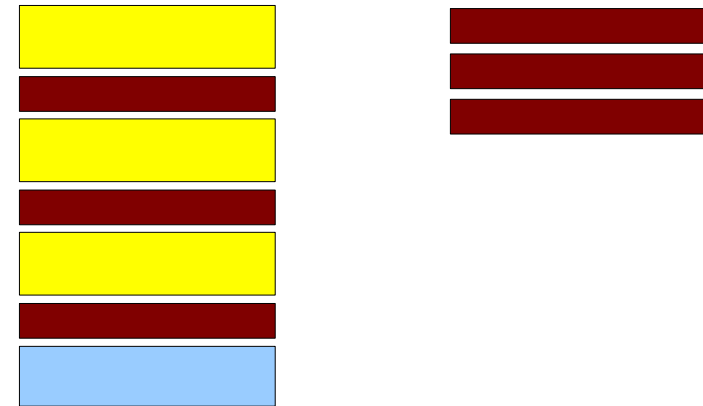


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space

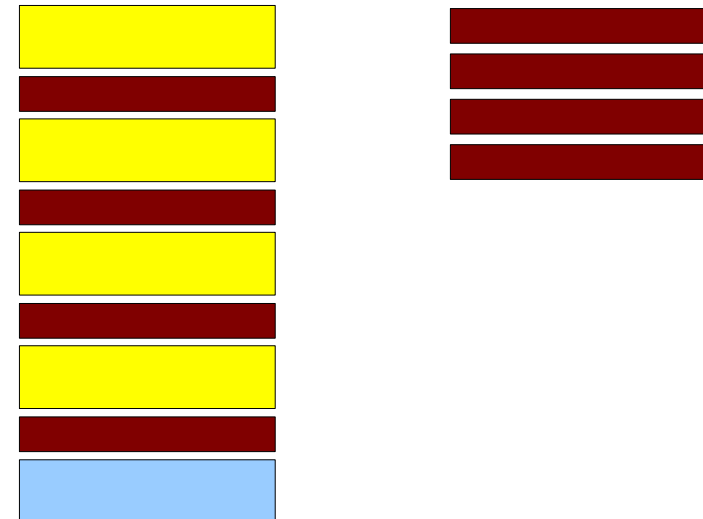


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space

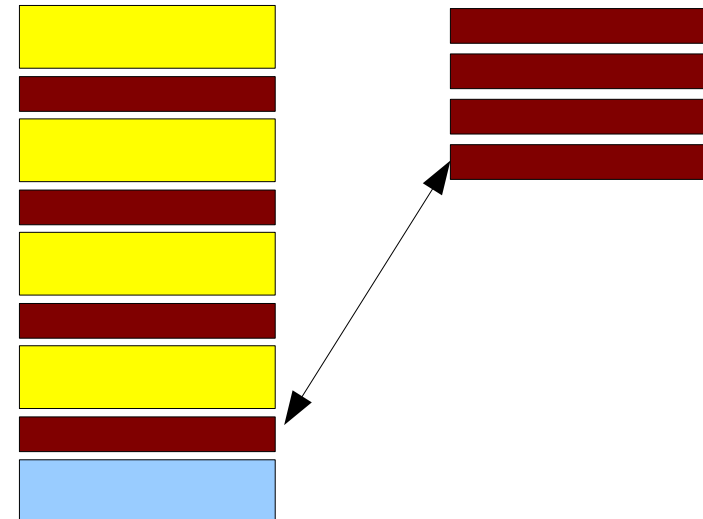


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space
- Check equality on `ret`

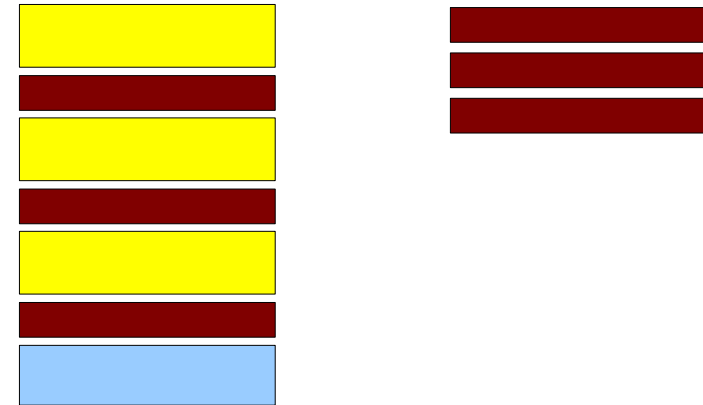


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space
- Check equality on `ret`

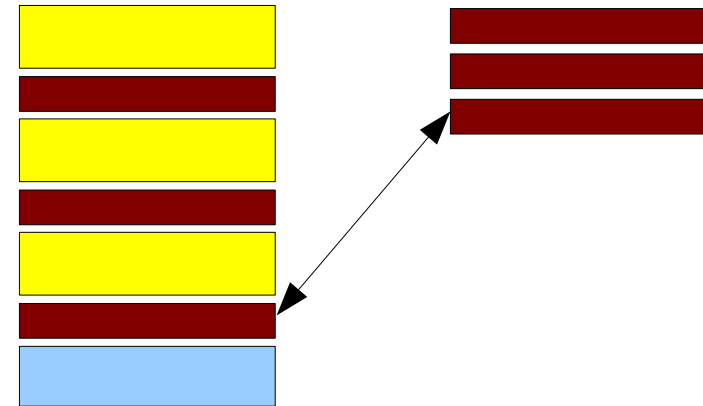


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space
- Check equality on `ret`

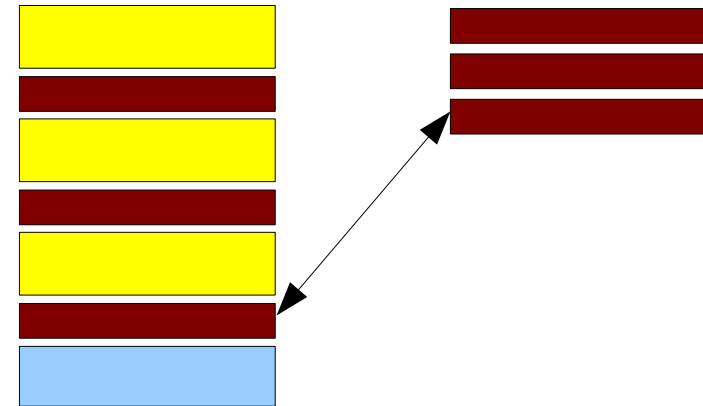


Stack protection

Control-Flow Protection

Shadow stack

- Replicates return addresses
- Somewhere else in address space
- Check equality on `ret`



Hardware support being added by Intel:
CET (Control-Flow Enforcement Technology)

- Shadow stacks only writable by `call/ret` instructions



Fortifying the build
`_FORTIFY_SOURCE`

`_FORTIFY_SOURCE`

`-O2 -D_FORTIFY_SOURCE=1` (or 2)

Includes various additional compile-time or run-time checks

- Array bounds
- Parameters
- Unused error result
- And more!

`__FORTIFY_SOURCE`

```
int main(void) {  
    char s[10];  
    read(STDIN_FILENO, s, 11);  
}
```

For `read`, `s` is just a `char *`, and it is told it is 11-bytes big.

But compiler knows better!

And libc headers can benefit from it

`__builtin_object_size(s, 0)` returns 10

abbreviated `__bos(s)`

`__FORTIFY_SOURCE`

In the libc **headers**

```
static inline
ssize_t read(int fd, void *buf, size_t n)
{
    return __read_chk(fd, buf, n, __bos(buf));
}
```

`__read_chk` can then check that $n \leq \text{__bos}(s)$

__FORTIFY_SOURCE

In the libc **headers**, even better:

```
static inline
ssize_t read(int fd, void *buf, size_t n)
{
    if (!__builtin_constant_p(n))
        return __read_chk(fd, buf, n, __bos(buf));
    if (n > __bos(buf))
        return __read_chk_warn(fd, buf, n, __bos(buf));
    return __read_alias(fd, buf, n);
}
```

__read_chk_warn wears a compile-time warning

_FORTIFY_SOURCE

```
int main(void) {  
    char s[10];  
    strcpy(s, "Hello, world!");  
}
```

Similar check for overflow

_FORTIFY_SOURCE

```
int main(void) {  
    int fd;  
    fd = open("test.txt", O_RDONLY);  
    ...  
}
```

_FORTIFY_SOURCE

```
int main(void) {  
    int fd;  
    fd = open("test.txt", O_RDWR|O_CREAT);  
    ...  
}
```

_FORTIFY_SOURCE

```
int main(void) {  
    int fd;  
    fd = open("test.txt", O_RDWR|O_CREAT, 0600);  
    ...  
}
```

Check for missing parameter

_FORTIFY_SOURCE

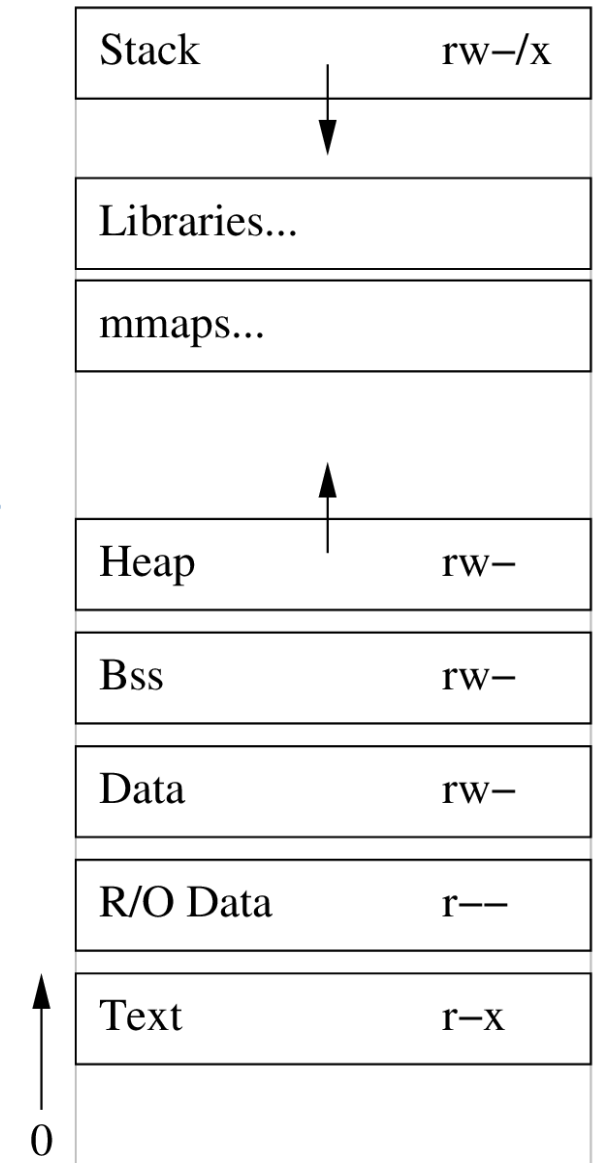
```
int main(int argc, char *argv[]) {  
    char *s;  
    asprintf(&s, "Hello, %s!\n", argv[1]);  
    ...  
}
```



Warn about unused result

_FORTIFY_SOURCE

```
int myfunc(const char *s) {  
    printf(s);  
    ...  
}
```

As discussed last week, bad idea.
But perhaps `s` really is a static constant string?
How to know?





Fortifying the build [almtu]san

Address / Leak / Memory / Thread / Undefined SANitizer

- In-compiler additions
- Small-ish checks
- Not negligible overhead! (can be 2x - 3x!)
- Very useful for debugging, Continuous Integration

Address / Leak SANitizer

- use-after-free

```
free(s); printf("%s\n", s);
```

- double-free

```
free(s); free(s);
```

- memory leaks

```
/* No free :) */
```

- use-after-return

```
int *f(void) {  
    int a;  
    return &a;  
}
```

- use-after-scope

```
int *p;  
{ int a; p = &a; }  
printf("%d\n", *p);
```

Address / Leak SANitizer

- heap/stack/global buffer overflow
 - Keeps track of set of valid addresses
 - Knows **exactly** where variables & arrays are!
 - Checks address on each pointer dereference

Memory SANitizer (LLVM-specific for now)

- Keeps track of set of **initialized** addresses
- Checks it on each pointer dereference for read

Thread SANitizer

- Looks out for
 - Race conditions
 - Lock ordering conflicts

Undefined SANitizer

- Looks out for undefined behavior (see previous course)
 - Integer overflow
 - Undefined integer shifts
 - ...

Cheat sheet

(bold options: now by default in Debian's dpkg-buildflags)

- -Wall -Wextra **-Wformat -Werror=format-security**
- **-fPIE -pie**
- **-Wl,-z,relro** -Wl,-z,now
- **-Wl,-z,noexecstack**
- **-fstack-protector-strong**
- -fstack-clash-protection
- **-D_FORTIFY_SOURCE=2 -O2**
- -fvtable-verify=std
- -fcf-protection=full

Use **hardening-check** to check your binaries

For debugging,

- -fsanitize=address (or leak, or memory, or thread, or undefined, or several at the same time)