String Oriented Programming: When ASLR is not enough

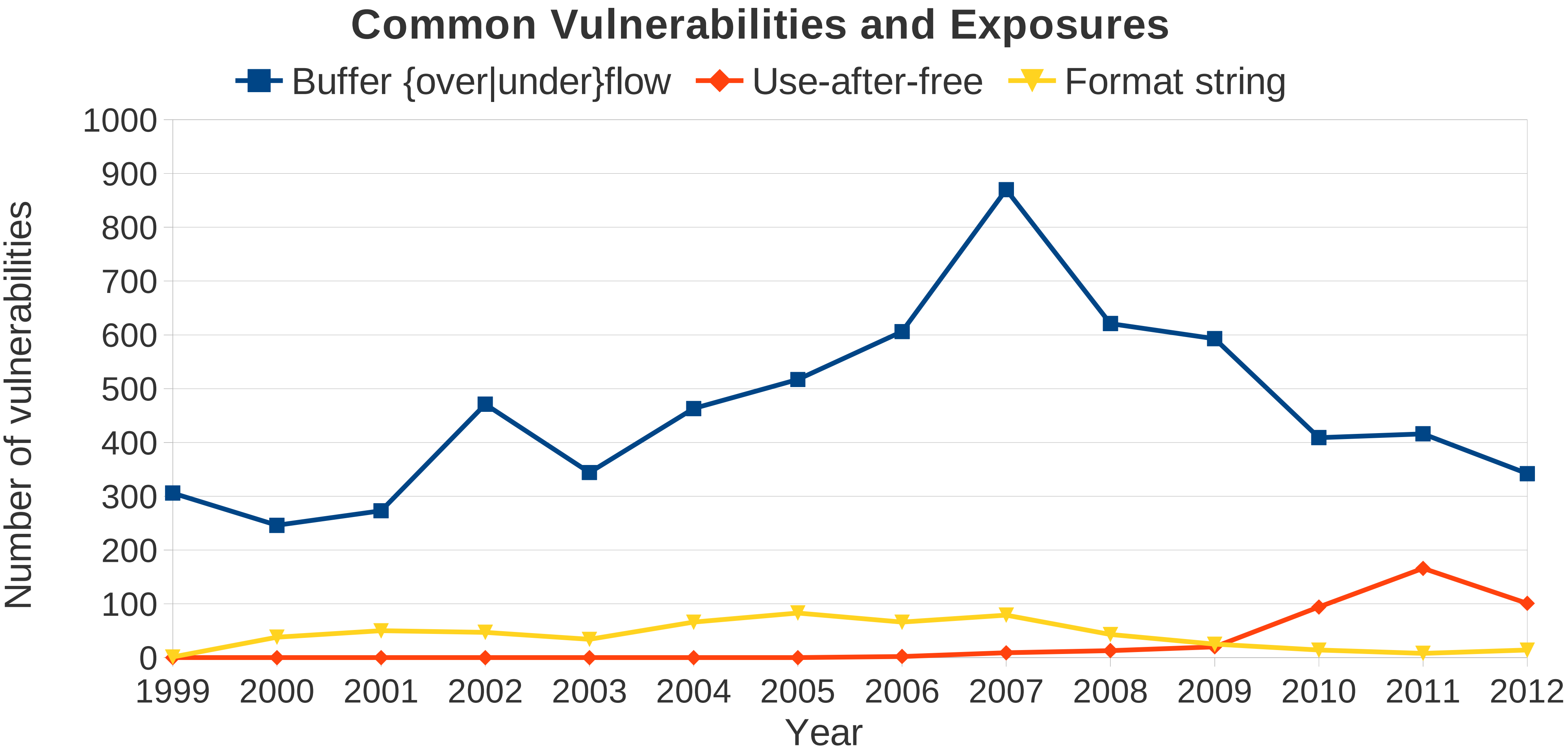
Mathias Payer* and Thomas R. Gross
Department of Computer Science
ETH Zürich, Switzerland

** now at UC Berkeley*

Current protection is not complete

http://creoflick.net

Motivation: circumvent protections



Format string exploits are often overlooked

- Drawback: hard to construct (due to protection mechanisms)
- Define a way to deterministically exploit format string bugs

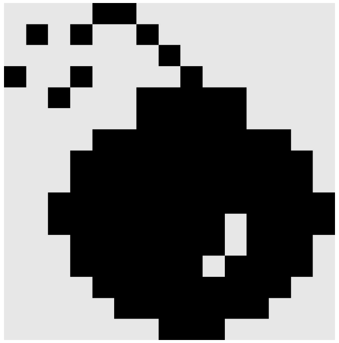
Attack model

Attacker with restricted privileges forces escalation

Attacker knows both source code and binary

Definition of a successful attack

- Redirect control flow to alternate location
- Injected code is executed or alternate data is used for existing code



Outline

Motivation

Attack model

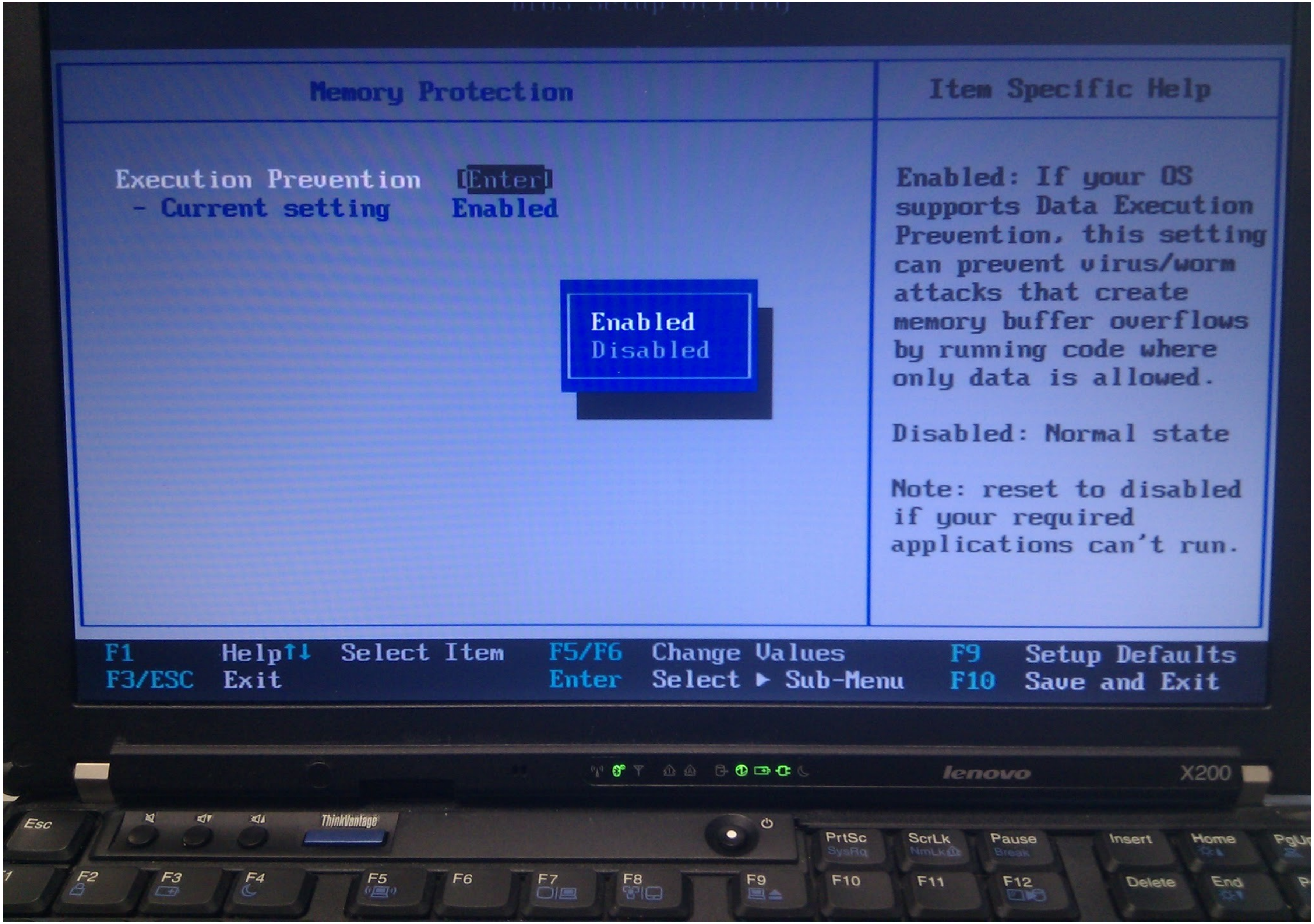
Current protection and their weaknesses

Attack building blocks

String Oriented Programming

Conclusion

Current protection



Memory Protection

```
Execution Prevention     [Enter]
 - Current setting       Enabled

                            Enabled
                            Disabled
```

Item Specific Help

Enabled: If your OS supports Data Execution Prevention, this setting can prevent virus/worm attacks that create memory buffer overflows by running code where only data is allowed.

Disabled: Normal state

Note: reset to disabled if your required applications can't run.

```
F1       Help↑↓  Select Item   F5/F6  Change Values        F9   Setup Defaults
F3/ESC   Exit                  Enter  Select ▶ Sub-Menu    F10  Save and Exit
```

Current protection

Data Execution Prevention

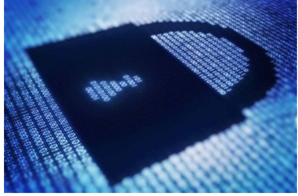
http://ww

Address Space Layout Randomization

Canaries

http://socialcanary.com

Data Execution Prevention (DEP)



DEP protects from code-injection attacks

- Based on page table modifications
- A memory page is either executable or writable (not both)

Weaknesses and limitations:

- No protection against code-reuse attacks like return-oriented programming or jump-oriented programming
- Self-modifying code not supported

Addr. Space Layout Rand. (ASLR)

ASLR randomizes code and data layout

- Probabilistic protection against attacks based on the loader
- Locations of all non-static memory regions are randomized during startup

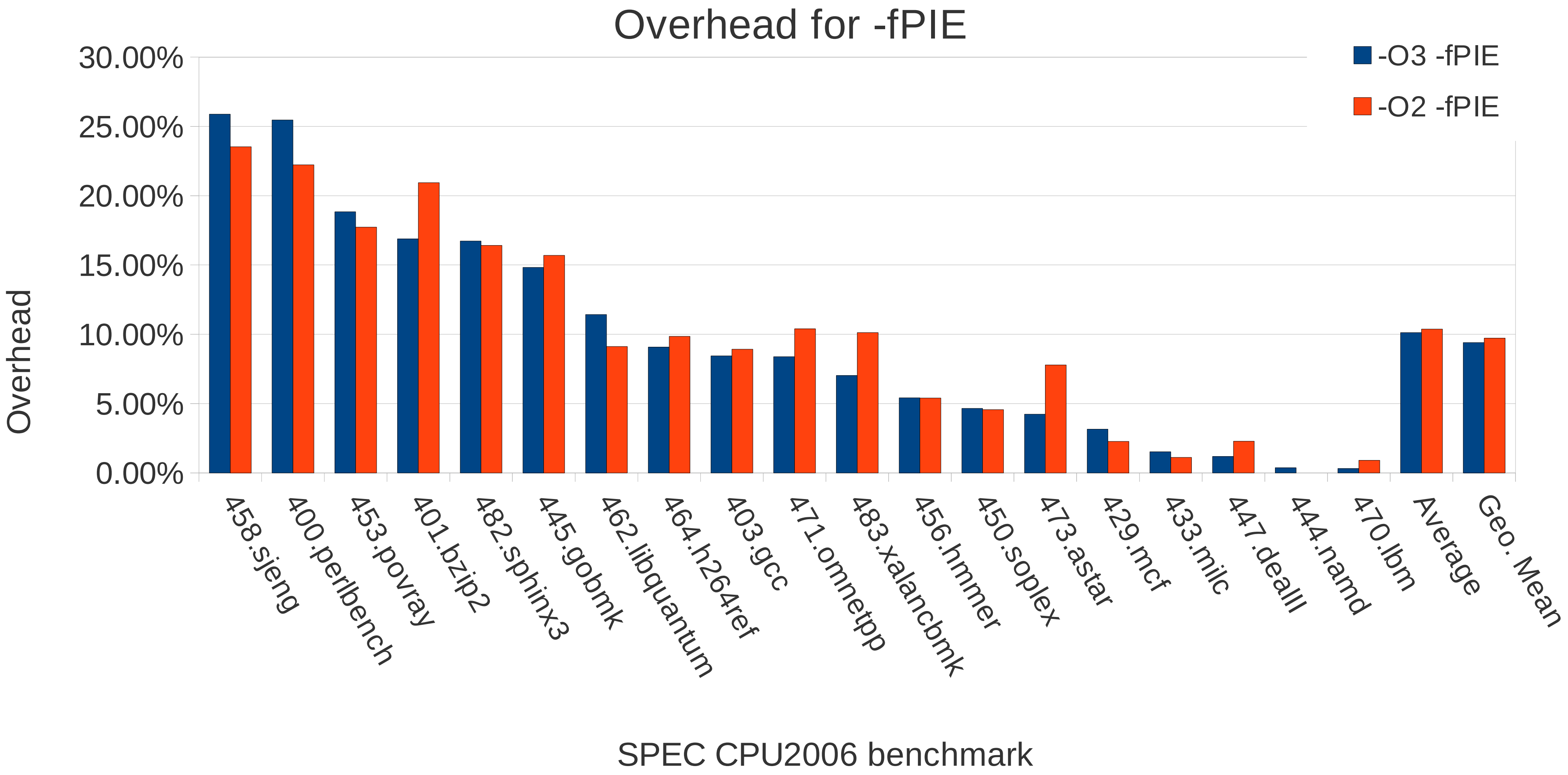
Weaknesses and limitations

- Some regions remain static for every run
- Prone to information leaks: randomization remains static during execution
- Performance impact on randomized code (~10%)

ASLR Performance Overhead

ASLR uses one register for PIC / ASLR code

- On IA32 this leads to a performance degradation



Canaries

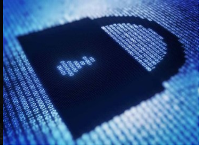
Canaries protect against buffer overflows

- Compiler modifies stack and structure layout
- Canaries are placed between buffers and other variables, content is verified after buffer operations

Weaknesses and limitations

- Only protect against continuous writes (buffer overflows)
- No protection against targeted writes (or reads)
- Prone to information leaks: usually one canary value per execution

Defense summary



Both Canaries and ASLR are probabilistic and prone to information leaks

- One shot attack becomes two shot attack
- Performance issues or data-layout issues

DEP gives us code integrity

- Protects against code injection, not code-reuse
- Code-reuse attacks needed to exploit DEP
- Hardware extension

Outline

Motivation

Attack model

Current protection and their weaknesses

Attack building blocks

String Oriented Programming

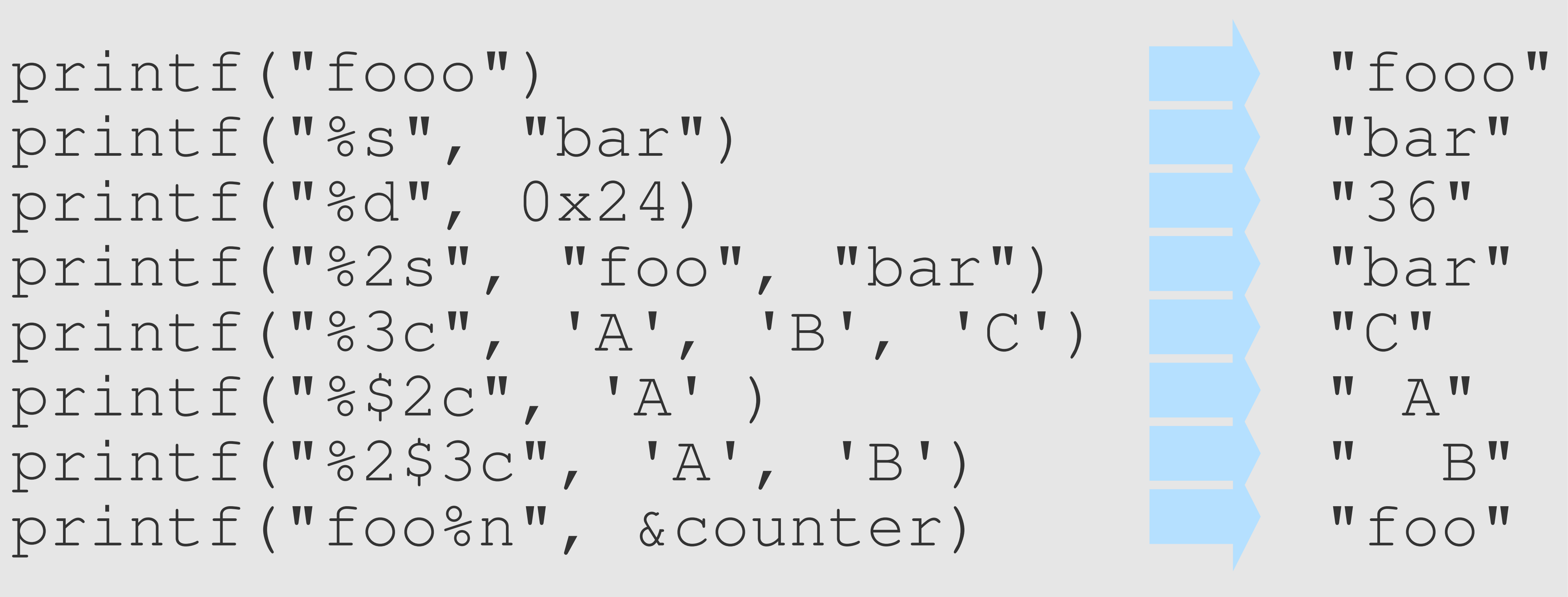
Mitigation

Conclusion

printf functionality

Format string contains tokens

- Each token consumes a stack slot and prints it
- Format dependent on token



```
printf("fooo")                                   "fooo"
printf("%s", "bar")                              "bar"
printf("%d", 0x24)                               "36"
printf("%2s", "foo", "bar")                      "bar"
printf("%3c", 'A', 'B', 'C')                     "C"
printf("%$2c", 'A' )                             " A"
printf("%2$3c", 'A', 'B')                        "  B"
printf("foo%n", &counter)                        "foo"
```

Format string attack*

Attacker controlled format results in arbitrary writes

- Format strings consume parameters on the stack
- `%n` token inverses order of input, results in indirect memory write
- Often string is on stack and can be used to store pointers

Write **`0xc0f3babe`** to **`0x41414141`**:

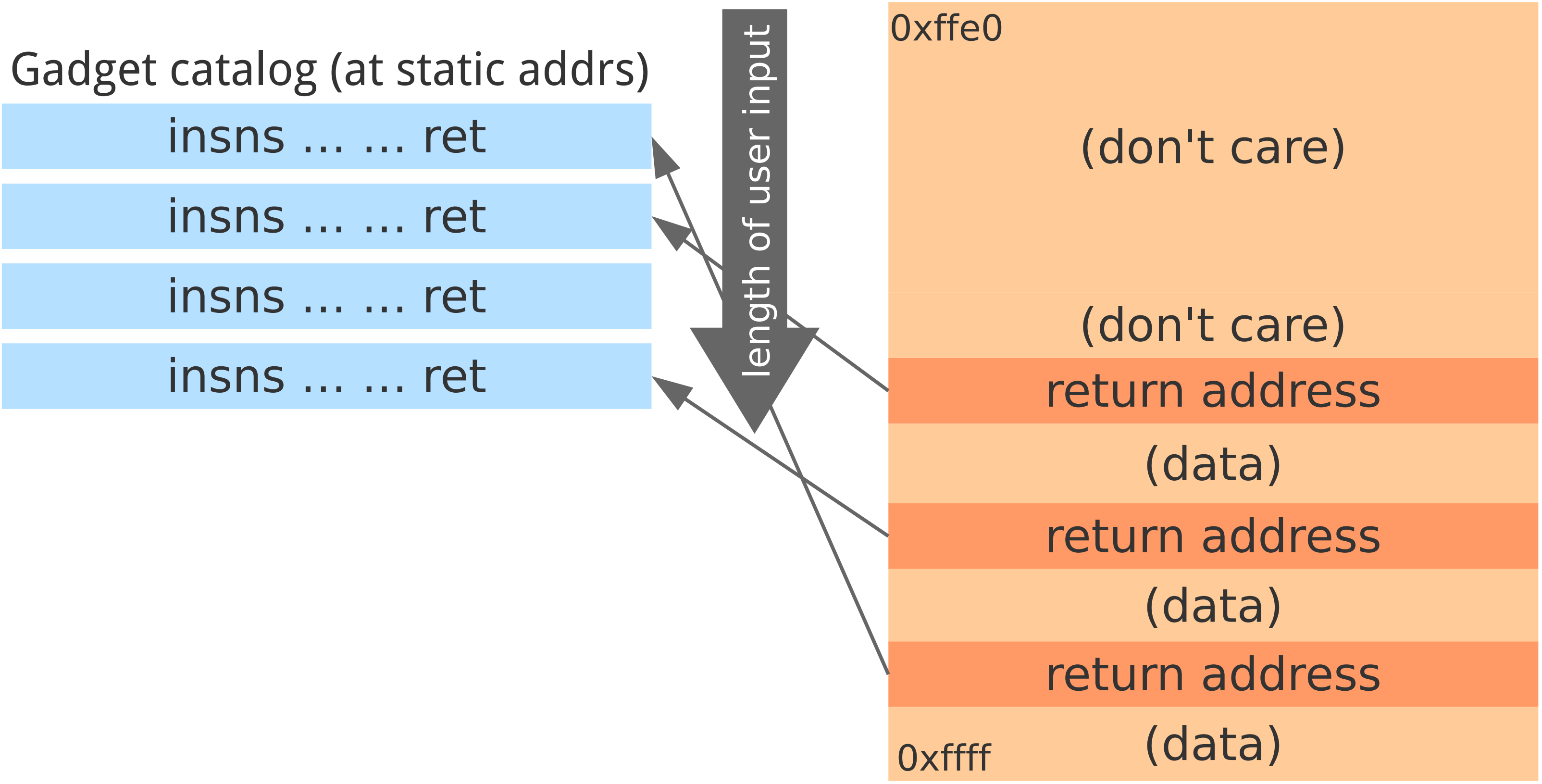
```
printf(
 "AAAACAAA"       /* encode 2 halfword pointers */
 "%1$49387c"      /* write 0xc0f3 - 8 bytes */
 "%6$hn"          /* store at second HW */
 "%1$63947c%5$hn" /* repeat with 0xbabe */
);
```

* many references, e.g., Haas, Defcon 18

Return Oriented Programming (ROP)*

ROP based on stack invocation frames

- Executes arbitrary code
- Initial bug prepares stack invocation frames



* Shacham, CCS'07

Outline

Motivation

Attack model

Current protection and their weaknesses

Format String Exploits

String Oriented Programming

- Technique
- Example

Mitigation

Conclusion

String Oriented Programming (SOP)

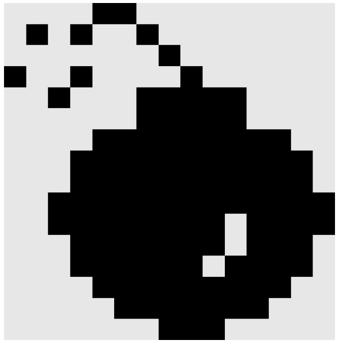
Observation: format string attacks inject data into running applications

Executing arbitrary code (through data)

- Needed: format string bug, attacker-controlled buffer on stack
- Not needed: buffer overflow, executable writable memory regions

SOP builds on ROP/JOP

- Overwrites static instruction pointers



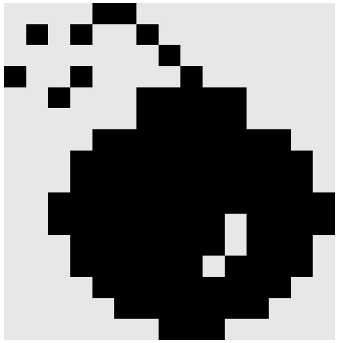
String Oriented Programming (SOP)

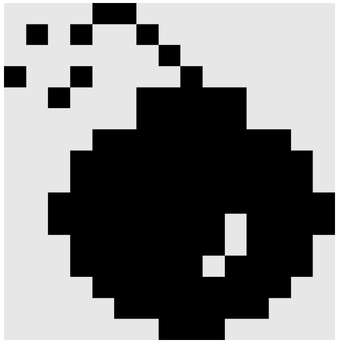
Patching and resolving addresses

- Application is static (this includes application's .plt and .got)
- Static program locations used to resolve relative addresses

Resolving hidden functions

- ASLR randomizes ~10bit for libraries
- Modify parts of static .got pointers
- Hidden functions can be called without loader support





Running example

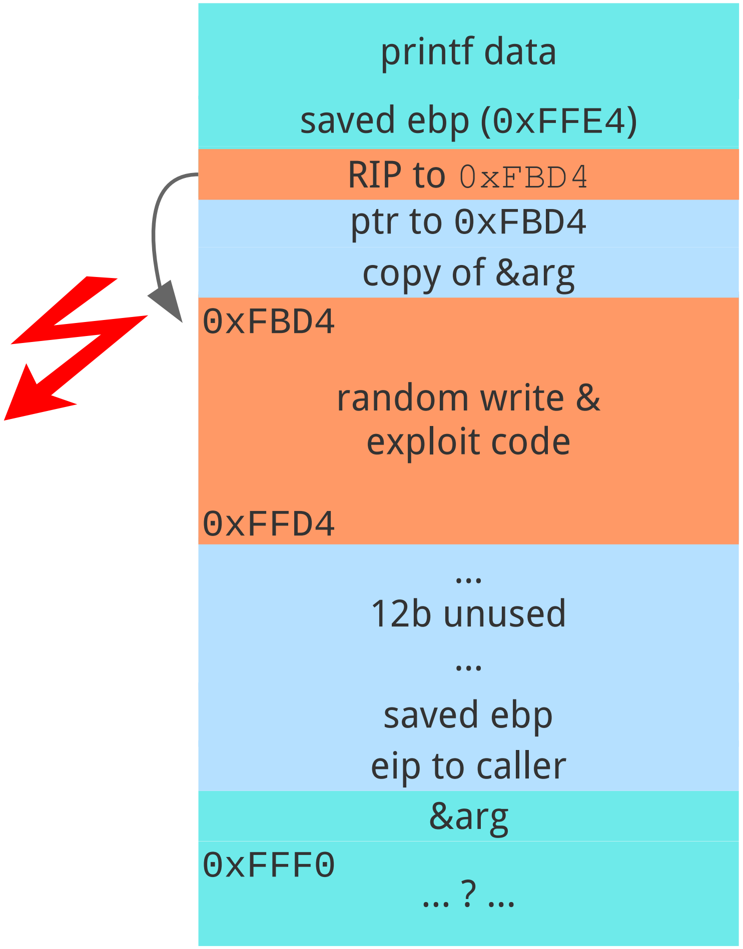
```
void foo(char *arg) {
  char text[1024];          // buffer on stack
  if (strlen(arg) >= 1024)  // length check
    return;
  strcpy(text, arg);
  printf(text);             // vulnerable printf
}

…

foo(user_str);              // unchecked user data

…
```

SOP: No Protection

All addresses are known, no execution protection, no stack protection

- Redirects control flow to code in the format string itself

```
void foo(char *arg) {
   char text[1024];
   if (strlen(arg) >= 1024)
      return;
   strcpy(text, arg);
   printf(text);
}

…

foo(user_str);

…
```



printf data

saved ebp (0xFFE4)

RIP to 0xFBD4

ptr to 0xFBD4

copy of &arg

0xFBD4

random write & exploit code

0xFFD4

…

12b unused

…

saved ebp

eip to caller

&arg

0xFFF0

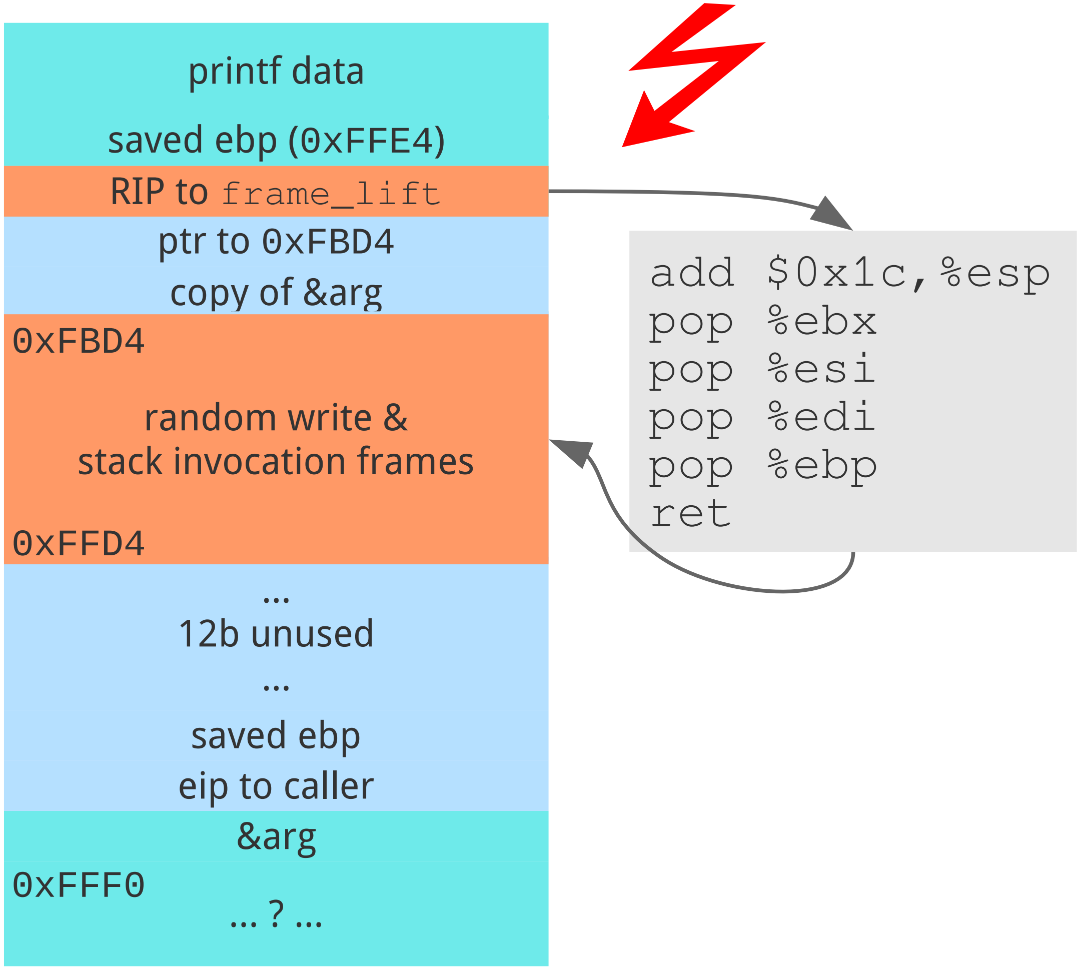
… ? …

SOP: Only DEP

DEP prevents code injection, rely on ROP/JOP instead

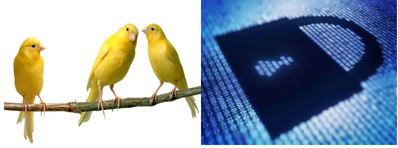
GNU C compiler adds `frame_lift` gadget

```
void foo(char *arg) {
  char text[1024];
  if (strlen(arg) >= 1024)
    return;
  strcpy(text, arg);
  printf(text);
}

…

foo(user_str);

…
```



```
add $0x1c,%esp
pop %ebx
pop %esi
pop %edi
pop %ebp
ret
```

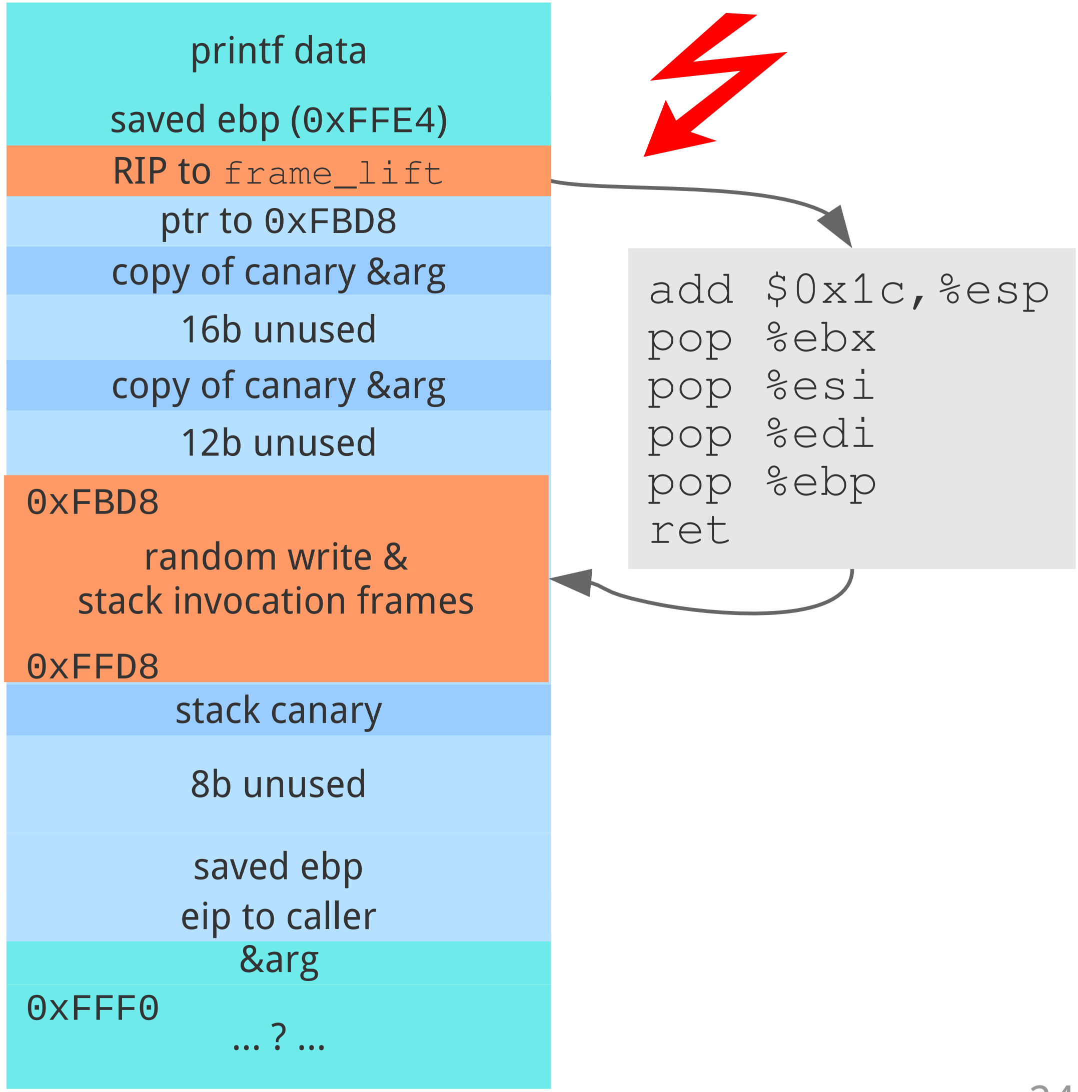
SOP: DEP & Canaries



ProPolice uses/enforces stack canaries

- Reuse attack mechanism, keep canaries intact

```
void foo(char *arg) {
  char text[1024];
  if (strlen(arg) >= 1024)
    return;
  strcpy(text, arg);
  printf(text);
}

…

foo(user_str);

…
```



printf data

saved ebp (0xFFE4)

RIP to `frame_lift`

ptr to 0xFBD8

copy of canary &arg

16b unused

copy of canary &arg

12b unused

0xFBD8

random write &
stack invocation frames

0xFFD8

stack canary

8b unused

saved ebp
eip to caller

&arg

0xFFF0
… ? …

```
add $0x1c,%esp
pop %ebx
pop %esi
pop %edi
pop %ebp
ret
```

SOP: ASLR, DEP, Canaries

Combined defenses force SOP to reuse existing code

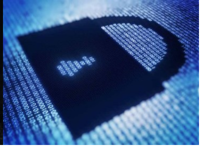
- Static code sequences in the application object
- Imported functions in the application (`.plt` and `.got`)

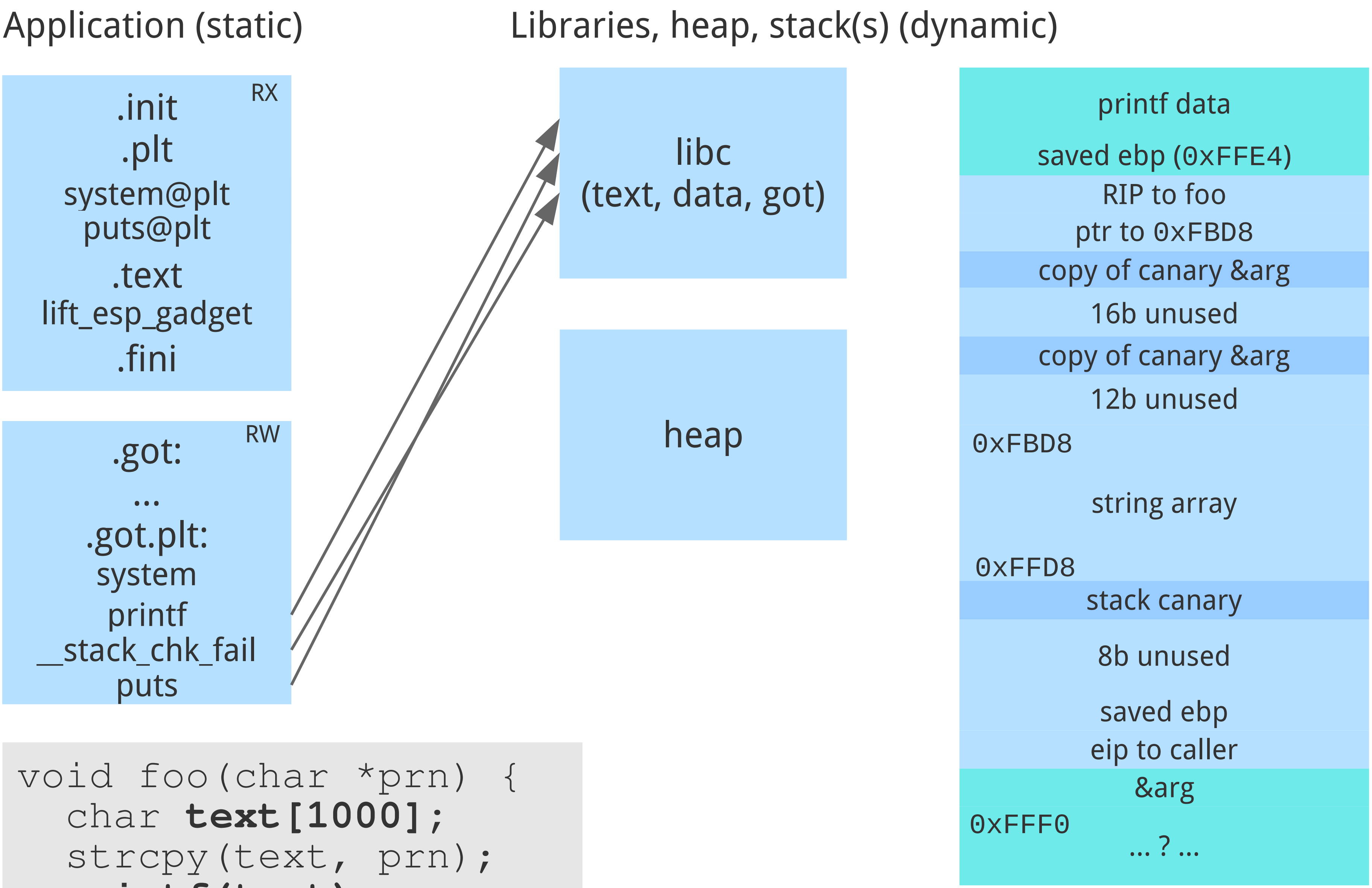
Use random byte-writes to adjust `.got` entries

- Enable other functions / gadgets that are not imported
- Combine stack invocation frames and indirect jump/call gadgets

```
void foo(char *prn)
{
  char text[1000];      // protected on stack
  strcpy(text, prn);
  printf(text);         // vulnerable printf
  puts("logged in\n"); // 'some' function
}
```

SOP: ASLR, DEP, Canaries





Application (static)

Libraries, heap, stack(s) (dynamic)

.init
.plt
system@plt
puts@plt
.text
lift_esp_gadget
.fini

RX

.got:
…
.got.plt:
system
printf
__stack_chk_fail
puts

RW

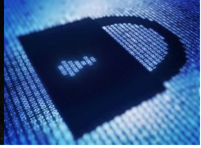
libc
(text, data, got)

heap

printf data
saved ebp (0xFFE4)
RIP to foo
ptr to 0xFBD8
copy of canary &arg
16b unused
copy of canary &arg
12b unused
0xFBD8
string array
0xFFD8
stack canary
8b unused
saved ebp
eip to caller
&arg
0xFFF0
… ? …

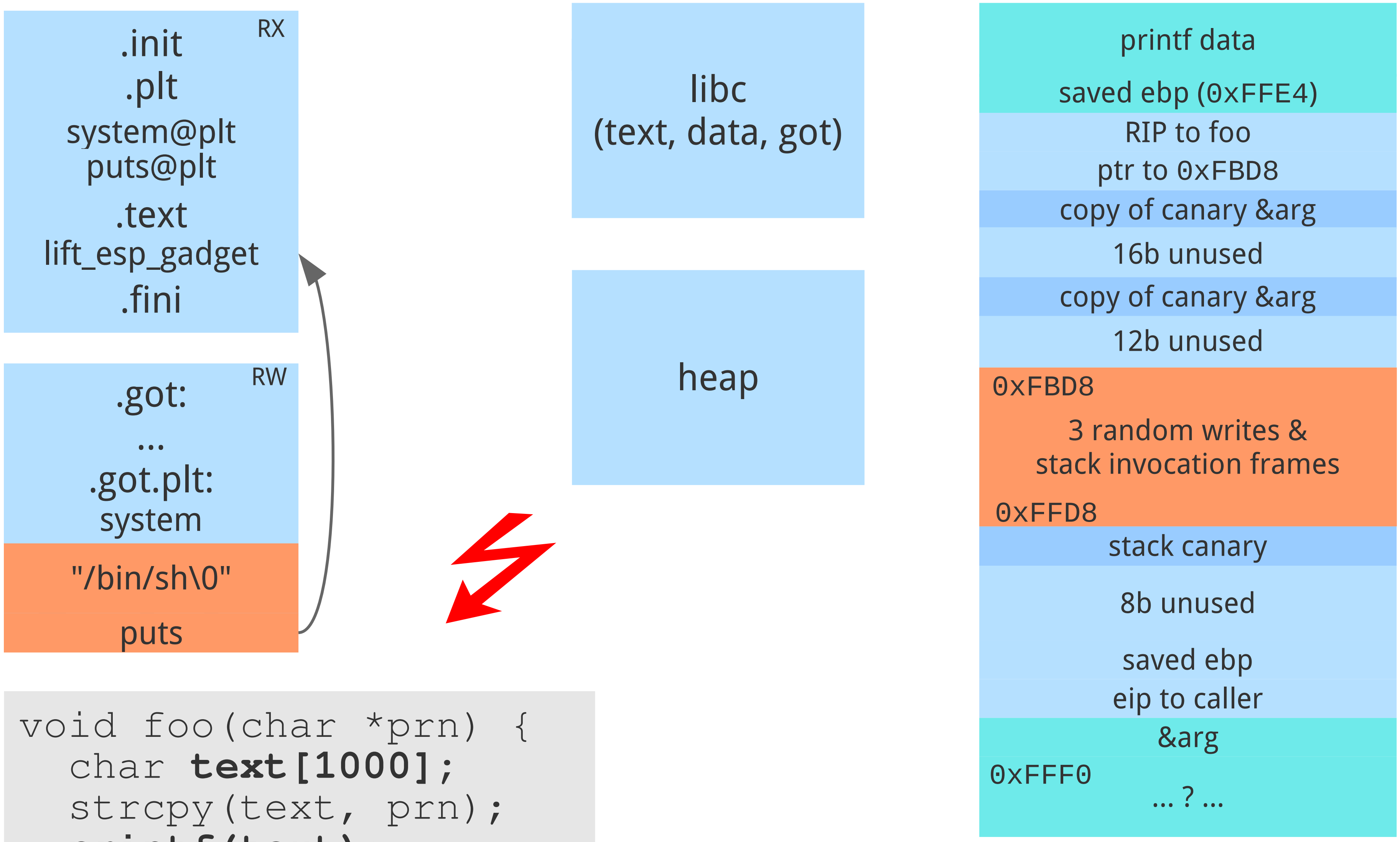
```
void foo(char *prn) {
  char text[1000];
  strcpy(text, prn);
  printf(text);
  puts("logged in\n");
}
```

SOP: ASLR, DEP, Canaries



Application (static)

Libraries, heap, stack(s) (dynamic)



RX

.init
.plt
system@plt
puts@plt
.text
lift_esp_gadget
.fini

RW

.got:
...
.got.plt:
system

"/bin/sh\0"

puts

libc
(text, data, got)

heap

printf data

saved ebp (0xFFE4)

RIP to foo
ptr to 0xFBD8

copy of canary &arg

16b unused

copy of canary &arg

12b unused

0xFBD8

3 random writes &
stack invocation frames

0xFFD8

stack canary

8b unused

saved ebp
eip to caller

&arg

0xFFF0

... ? ...

```
void foo(char *prn) {
  char text[1000];
  strcpy(text, prn);
  printf(text);
  puts("logged in\n");
}
```

Outline

Motivation

Attack model

Current protection and their weaknesses

Format String Exploits

String Oriented Programming

Mitigation

Conclusion

Mitigation

Control-flow protection

- Use a shadow stack to protect the RIP
- Protect indirect control flow transfers

Disable writes in format strings

- Remove `%n` processing
- Add (static and dynamic) compiler checks for valid targets

Outline

Motivation

Attack model

Current protection and their weaknesses

Format String Exploits

String Oriented Programming

Mitigation

Conclusion

Conclusion

String Oriented Programming (SOP)

- Based on format string exploit
- Extends code-reuse attacks (ROP / JOP)
- Naturally circumvents DEP and Canaries
- Reconstructs pointers and circumvents ASLR

Format string bugs result in complete compromise of the application and full control for the attacker

- SOP protection needs more work (virtualization, or secure libc?)
- Look at the complete toolchain

Questions?

?

Other protection mechanisms

Stack integrity (StackGuard, Propolice)

Verify library usage (Libsafe / Libverify)

Pointer encryption (PointGuard)

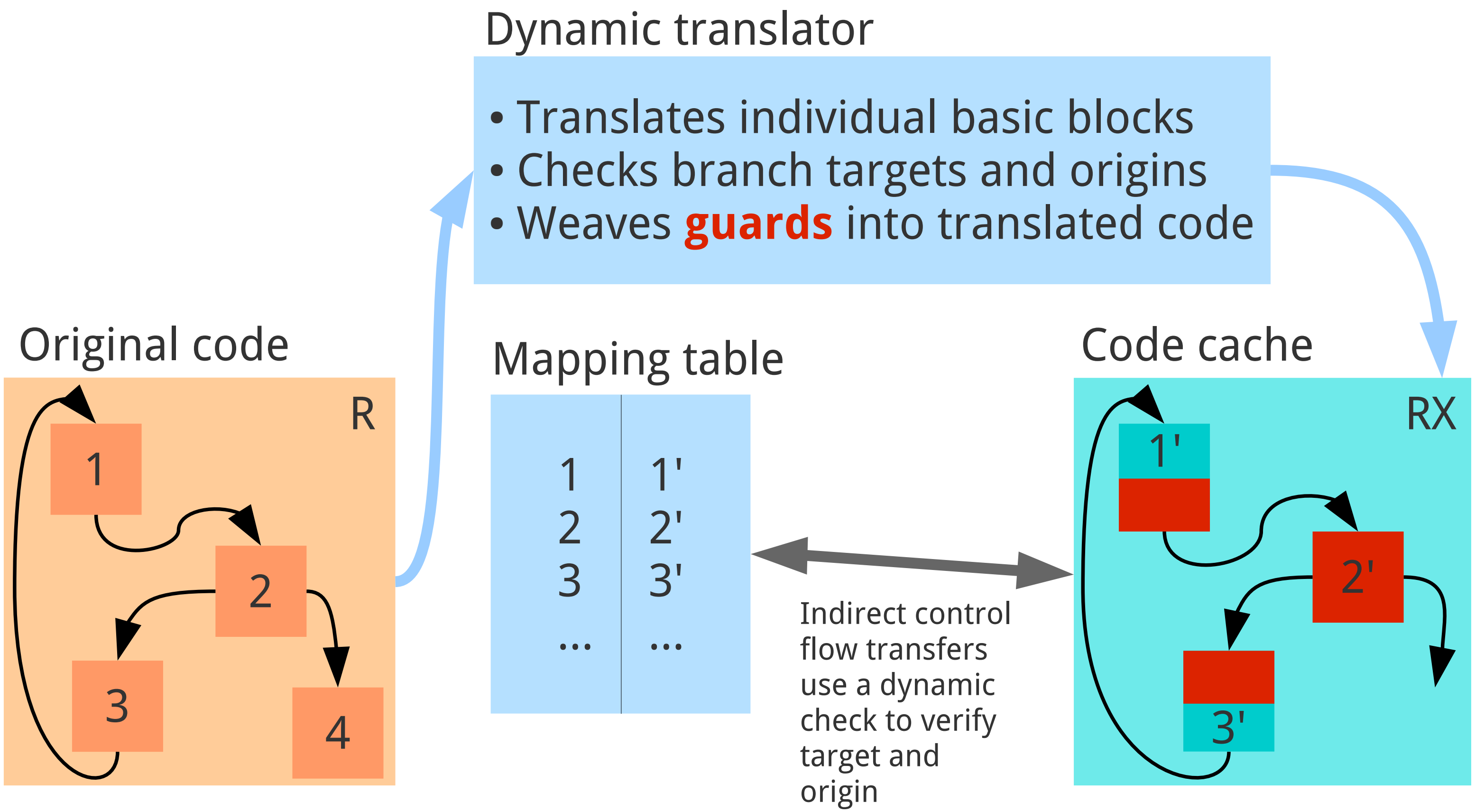
ISA modifications (ISA randomization)

Format string protection (FormatGuard)

Randomize memory locations (ASLR)

Check/verify control flow transfer (CFI / XFI)

Software based fault isolation



Dynamic translator

- Translates individual basic blocks
- Checks branch targets and origins
- Weaves **guards** into translated code

Original code

Mapping table

1	1'
2	2'
3	3'
…	…

Indirect control flow transfers use a dynamic check to verify target and origin

Code cache