Automatic Discovery of API-Level Exploits

Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha,
Thomas W. Reps and Randal E. Bryant

University of Wisconsin and Carnegie Mellon University

To appear at 27th Intl. Conf. on Software Engineering
St. Louis, Missouri; May 2005

Two definitions

Vulnerability

An error in a software package that allows for unintended behavior.

Exploit

A sequence of operations that attacks the vulnerability, typically with malicious intent.

Motivation

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```

- Format-string vulnerability
  - `buf = “%s%s%s”`
  - `fprintf(LOG,“%s%s%s”)`

- Insufficient arguments to `fprintf.`Possible outcomes
  - Unintelligible log entry.
  - Program crash.
  - Hacker takes over program!

Motivation

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```

- Tools to find format-string vulnerabilities: Percent-S [Shankar et al. USENIX Security 2001]
  - Finds user-controlled format-strings (using type-qualifiers)
- But, tools to *systematically* find *exploits* against such vulnerabilities?
- Trend is similar for other kinds of vulnerabilities.

Motivation

Many vulnerability-detection tools. Few, if any, exploit-finding tools.

Q: What is different about exploit-finding?

Q: Is exploit finding worth the effort?

Q: Isn’t finding exploits a black-hat activity?

Motivation

Many vulnerability-detection tools. Few, if any, exploit-finding tools.

Q: What is different about exploit-finding?

A: Modeling low-level implementation details.

Q: Is exploit finding worth the effort?

A: Yes!

Q: Isn’t finding exploits a black-hat activity?

A: Not necessarily!

Exploit-finding can benefit, and improve the quality of, vulnerability-detection tools.

Overview of results

- We study exploit-finding by considering a class of exploits called *API-Level Exploits.*

- We present a framework to:
  - Model low-level details of an API’s implementation.
  - Automatically analyze the model and find exploits.

- Two real-world instantiations:
  - **`printf`**-family format-string exploits.
  - IBM Common Cryptographic Architecture (CCA) API.

Talk structure

- Motivation and Overview.
- Framework for finding API-level exploits.
- Example: format-string exploit-detector.
  - Overview of **`printf`** and format-string exploits.
  - Instantiating **`printf`** in our framework.
  - Results.
  - Comparison with other tools.
- Related work.
- Conclusions.

API-Level Exploits

- What are API-Level exploits?
  - A sequence of API operations *allowed* by the underlying system.
  - But, compromises the security of the system.

- Example: [Chen and Wagner, CCS 2002]
  - System: UNIX, API: system calls.
  - **`setuid(0)`** followed by **`execl`** can lead to **`root`** privileges.

Framework for modeling APIs

- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
  - Example: OS, system calls. Want to check if a particular application can compromise the OS.
  - Only check sequences of system calls generated by that application [Giffin et al. NDSS 2004]

Framework for modeling APIs

- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
- Model system S as:

$$\boxed{S = (V, \mathrm{Init}, \Sigma, L)}$$

Framework for modeling APIs

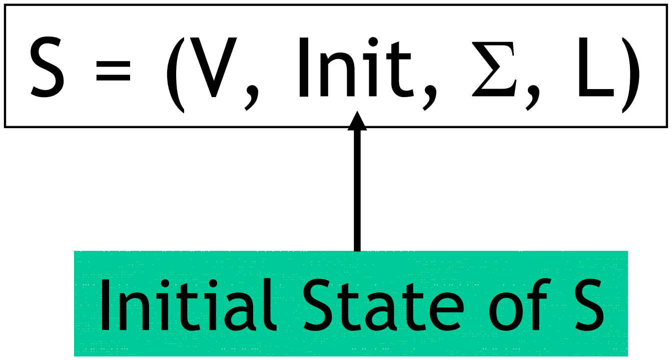
- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
- Model system S as:

$$\boxed{S = (V, \text{Init}, \Sigma, L)}$$

Finite set of variables, denoting current state of S.
Possibly with values from an infinite domain

Framework for modeling APIs

- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
- Model system S as:



$$\boxed{S = (V, \mathit{Init}, \Sigma, L)}$$

Initial State of S

Framework for modeling APIs

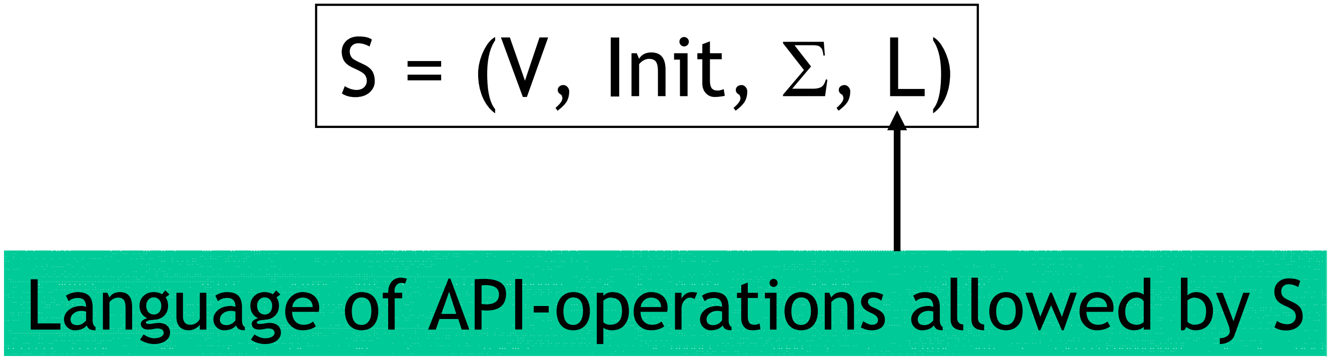
- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
- Model system S as:

$$\boxed{S = (V, \text{Init}, \Sigma, L)}$$

Finite set of API operations. Semantics of each operation specified using Pre- and Post-contitions

Framework for modeling APIs

- Find exploits:
  - Model low-level details of the system.
- Only check allowed sequences:
  - Otherwise, false alarms.
  - Must encode sets of allowed sequences.
- Model system S as:



$$S = (V, \mathrm{Init}, \Sigma, L)$$

Language of API-operations allowed by S

Finding API-Level Exploits

- Specify what is Bad for the system S.
- Reduce to satisfiability.
- Is there a sequence of k operations, such that
  - For any finite value of k,
  - S initially satisfies predicate Init,
  - The sequence of operations is in L,
  - The state of S after the k^{th} operation satisfies Bad

Finding API-Level Exploits

- Specify what is Bad for the system S.
- Reduce to satisfiability.
- Is there a sequence of k operations, such that
  - For any finite value of k,
  - S initially satisfies predicate Init,
  - The sequence of operations is in L,
  - The state of S after the kth operation satisfies Bad
- Not surprisingly, undecidable.
  - k is unbounded.
  - In general, system is infinite-state.

Finding API-Level Exploits

- Our approach:
  - Bound k, the length of the sequence of API operations.
  - Model check.
- In effect, checking all allowed sequences of length k for exploits.

Talk structure

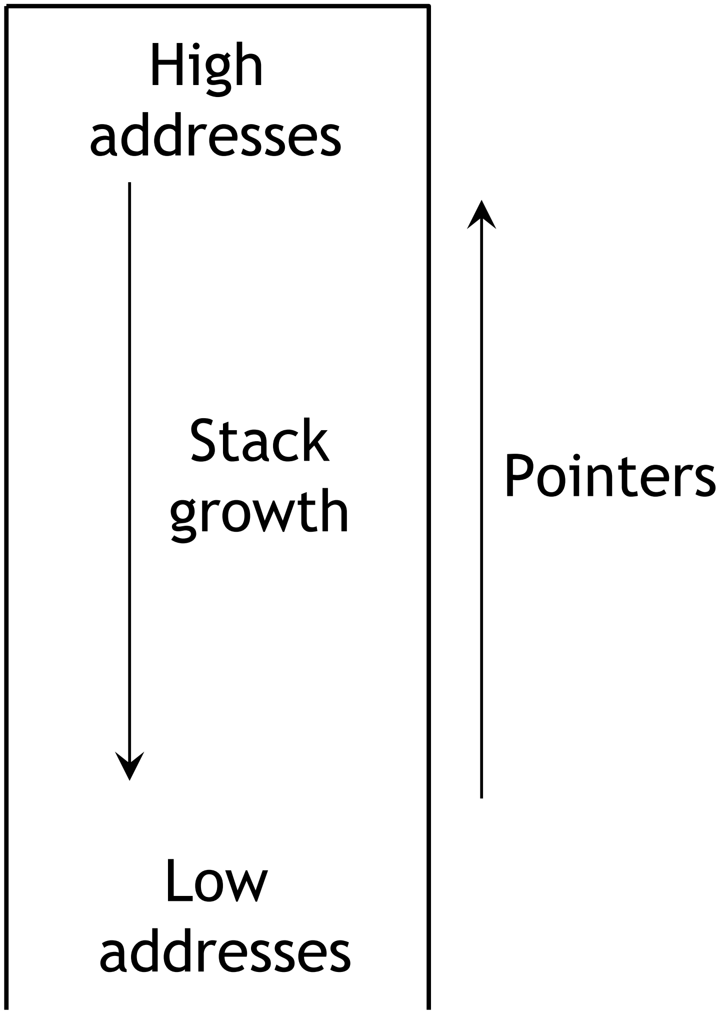
- Motivation and Overview.
- Framework for finding API-level exploits.
- Example: format-string exploit-detector.
  - Overview of **`printf`** and format-string exploits.
  - Instantiating **`printf`** in our framework.
  - Results.
  - Comparison with other tools.
- Related work.
- Conclusions.

Format-string vulnerabilities

- Allow intruder to assume privileges of the victim program.
- Highly prevalent. [http://www.securiteam.com/exploits]
- Vulnerability-detection tools available.
  - Example: Percent-S.
- Goals of our tool:
  - Systematically find exploits against such vulnerabilities.
  - Work with real-world applications.

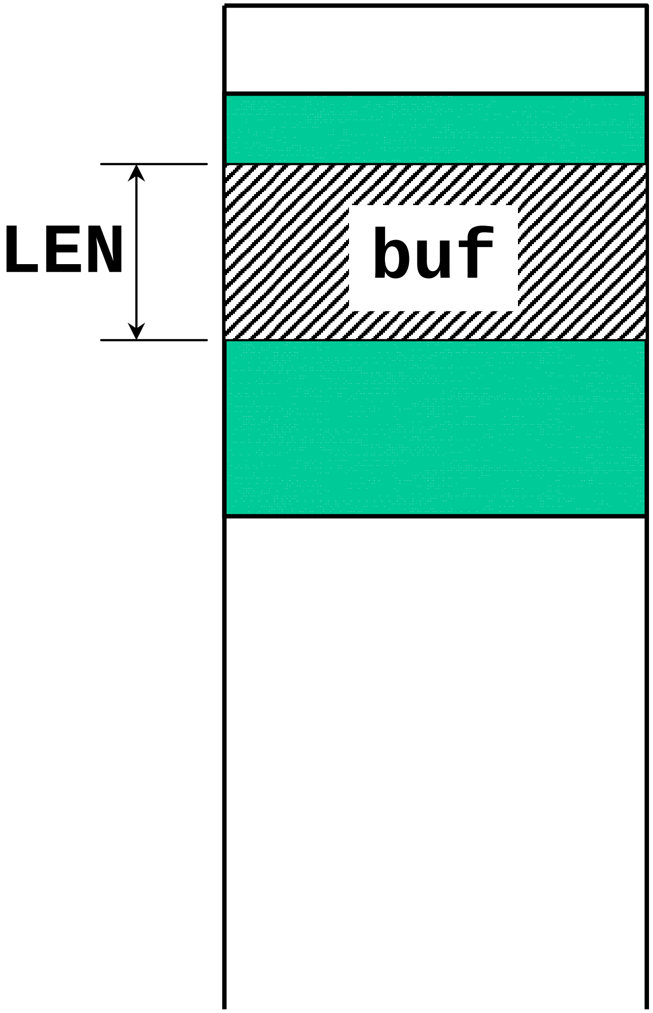
Overview of **`printf`**

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



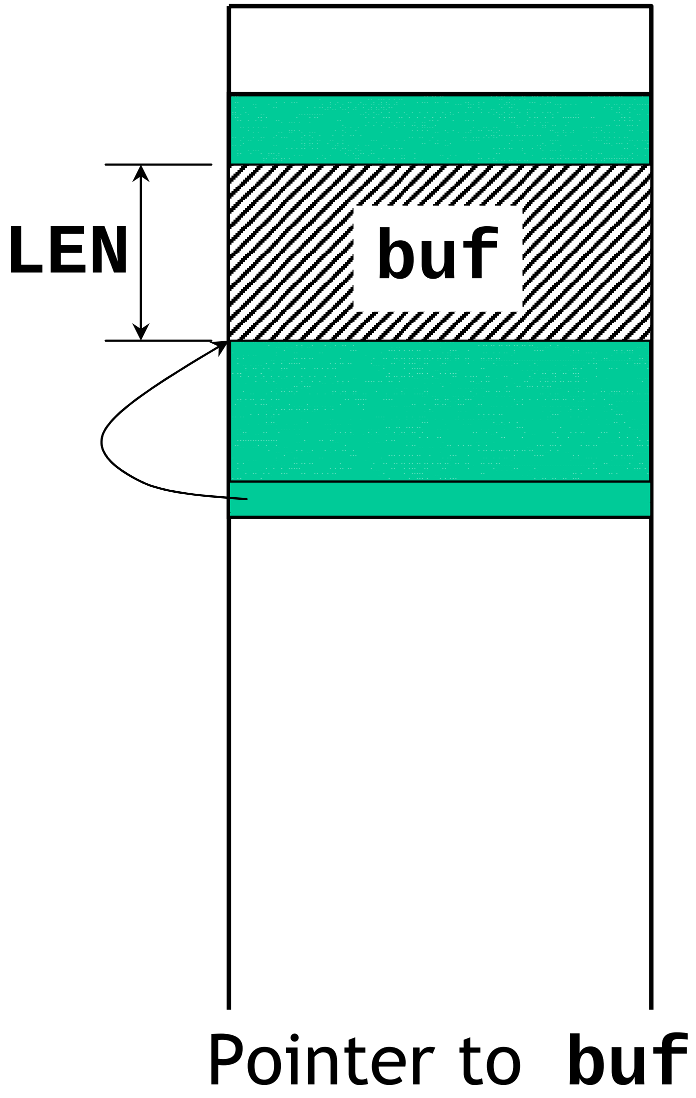
Overview of **`printf`**

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



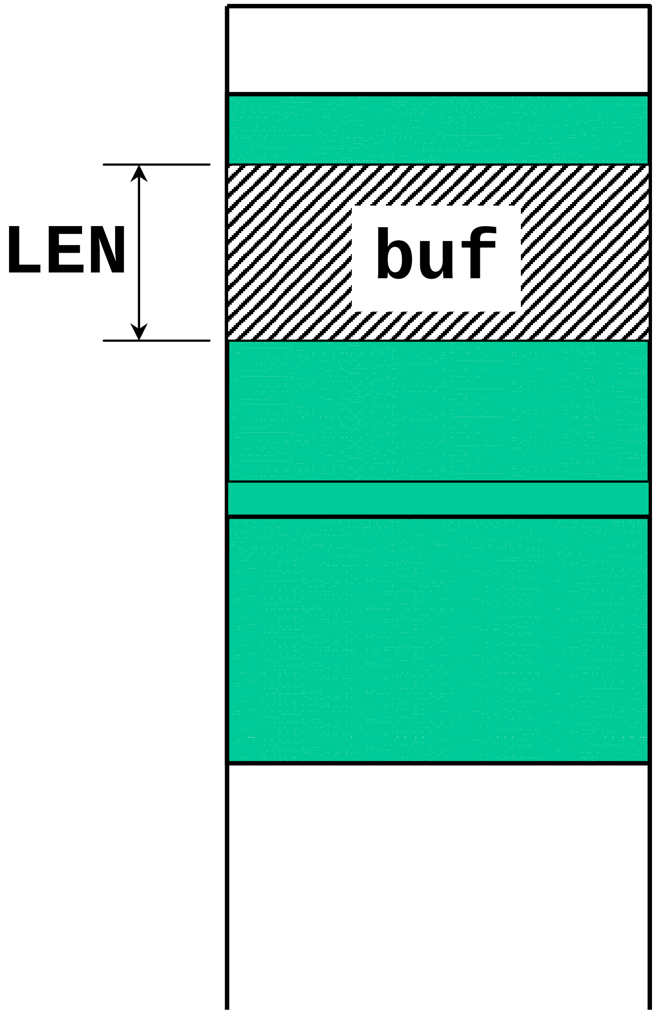
Overview of **printf**

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



Overview of `printf`

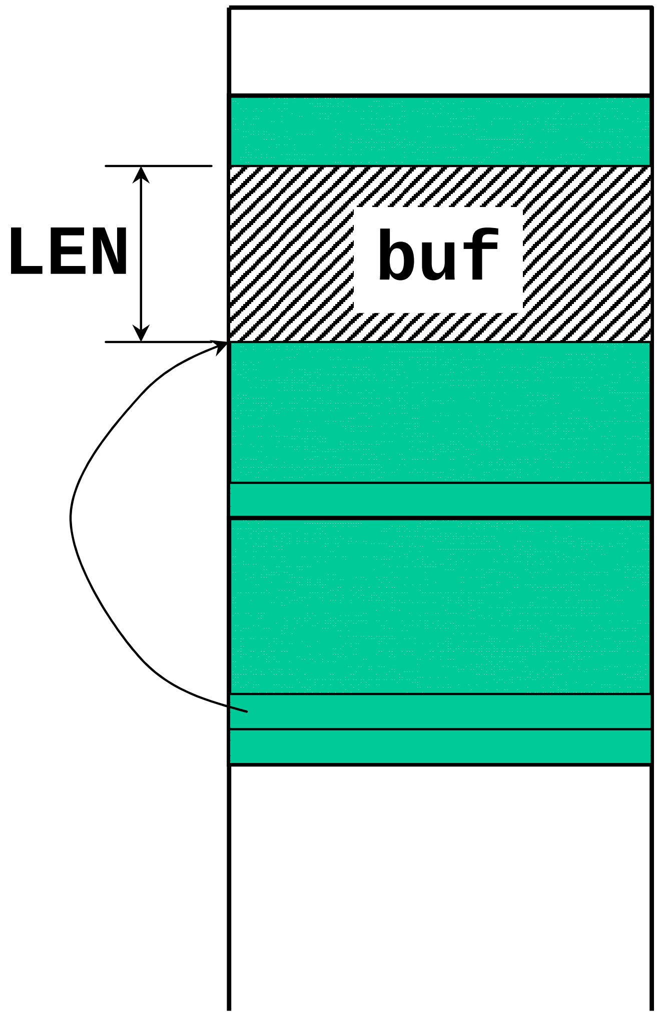
```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



Stack frame of `log`

Overview of **printf**

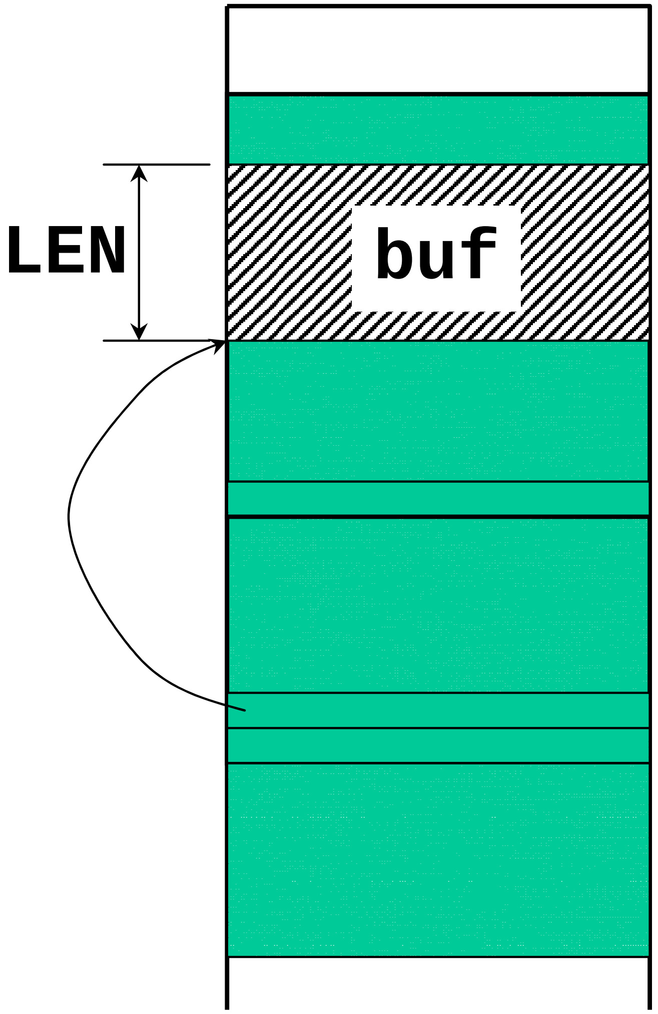
```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



Pointer to **buf**

Overview of **printf**

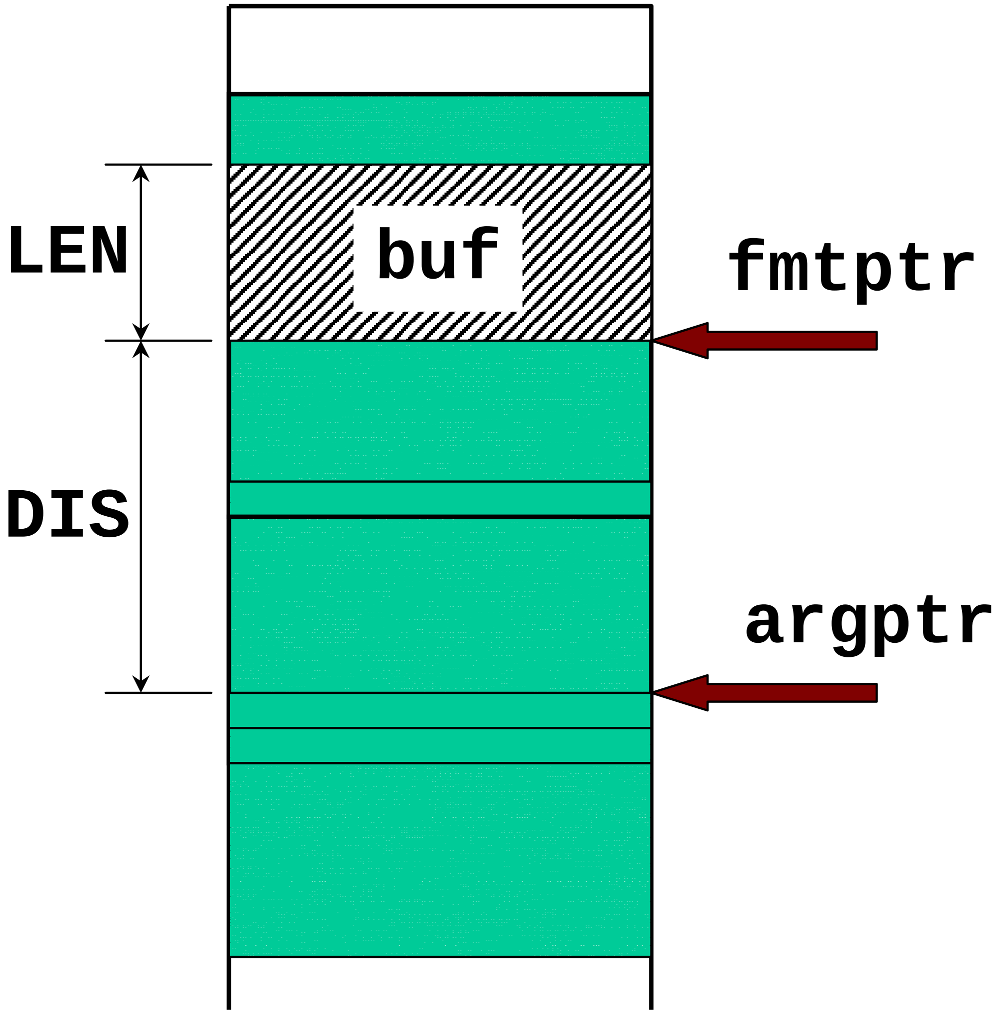
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Stack frame of **fprintf**

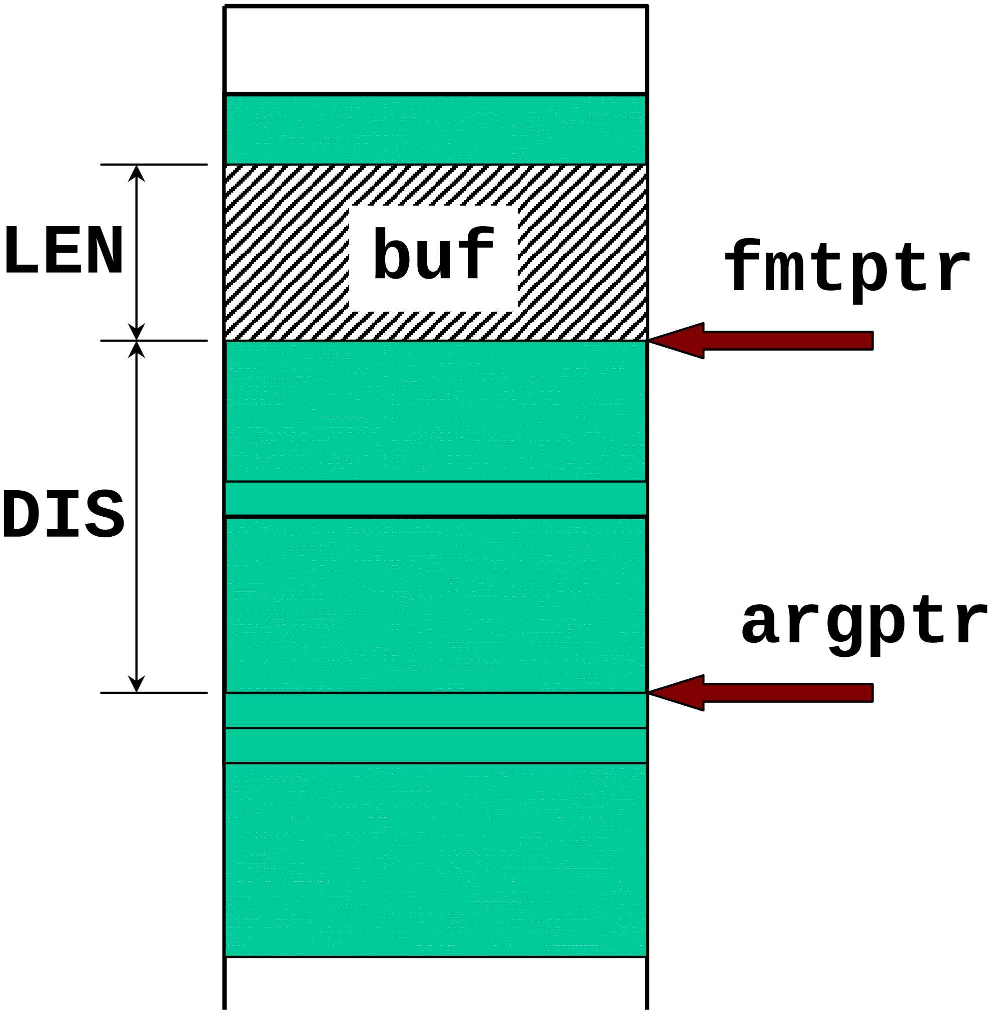
Overview of **`printf`**

```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



Overview of **`printf`**

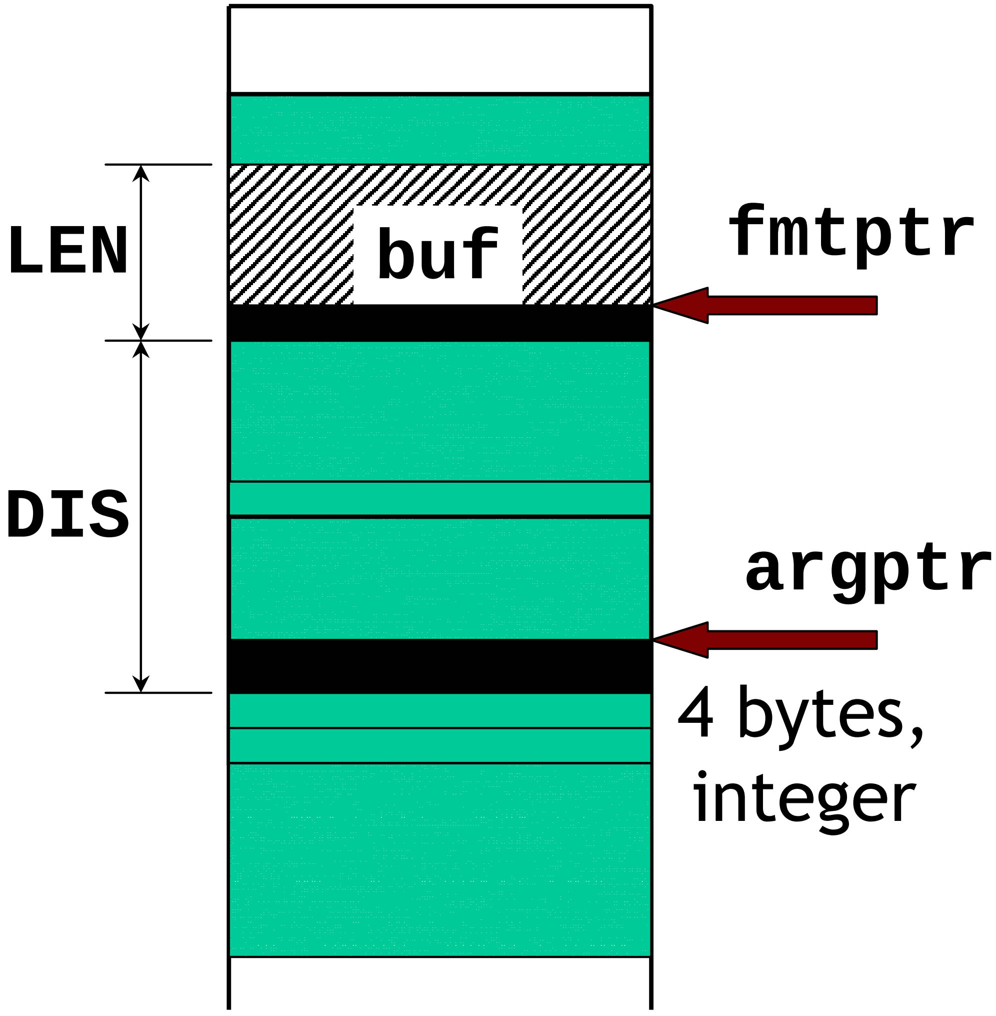
```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



buf = “%x%x%s”

Overview of **printf**

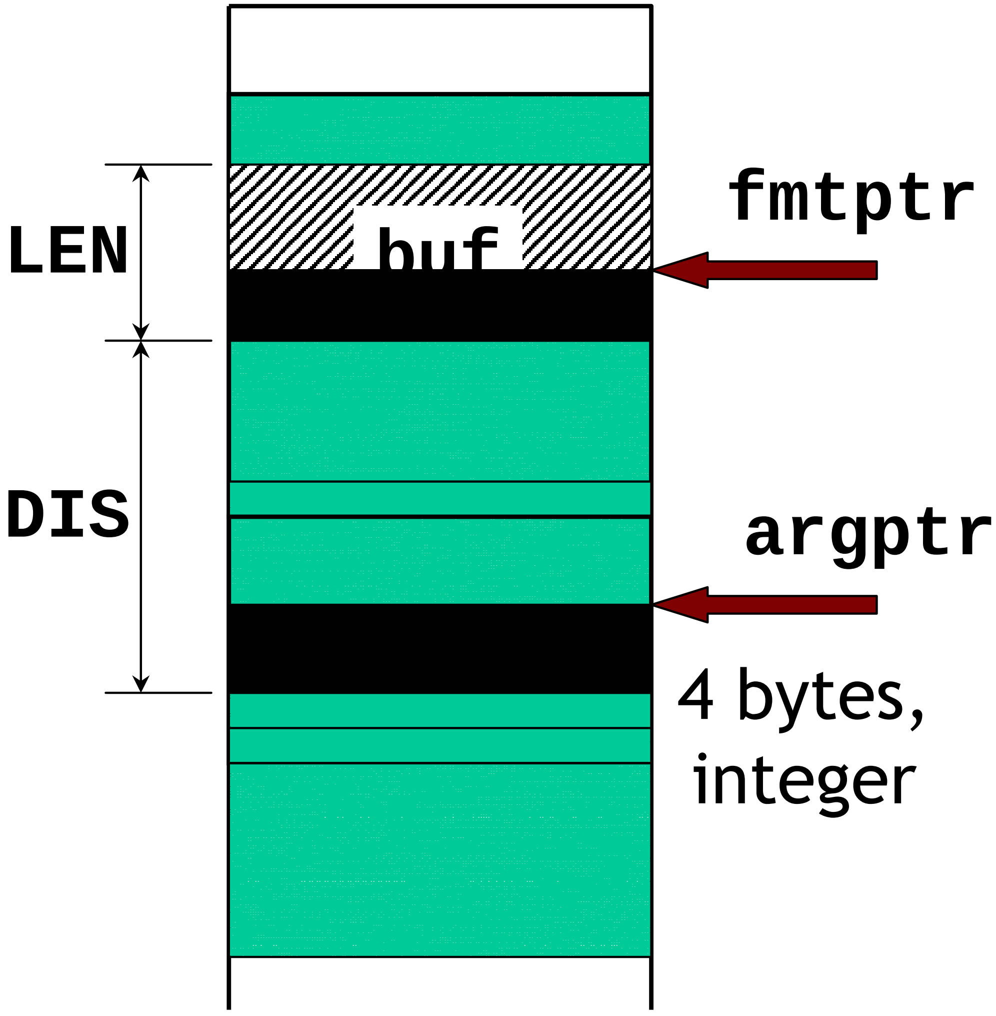
```
//Format & enter into LOG
void log(char *fmt,...){
   fprintf(LOG,fmt,...);
   return;
}

//Call log on user input
int foo(void){
   char buf[LEN];
   …
   fgets(buf,LEN-1,FILE);
   log(buf);
   …
}
```



buf = “🟥%x%s”

Overview of printf

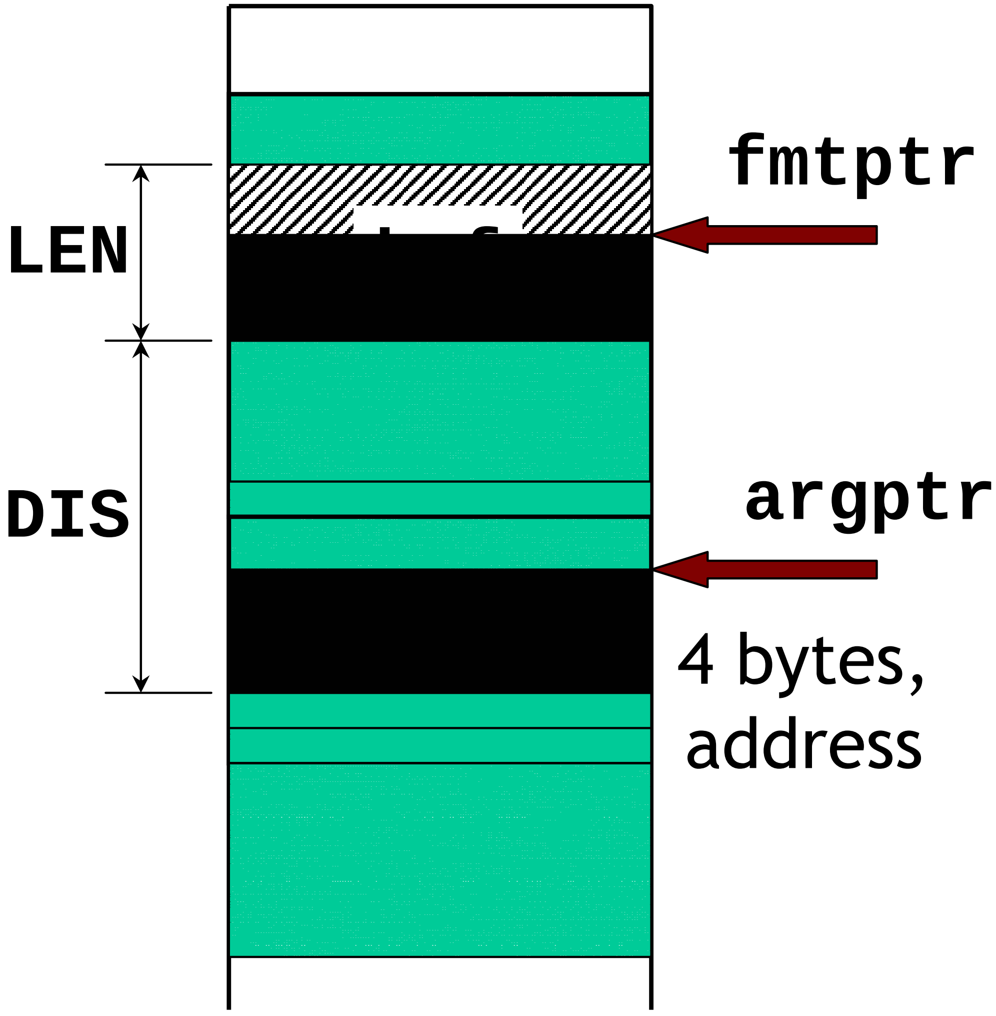
```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```



buf = “🟥%s”

Overview of **printf**

```
//Format & enter into LOG
void log(char *fmt,...){
  fprintf(LOG,fmt,...);
  return;
}

//Call log on user input
int foo(void){
  char buf[LEN];
  …
  fgets(buf,LEN-1,FILE);
  log(buf);
  …
}
```

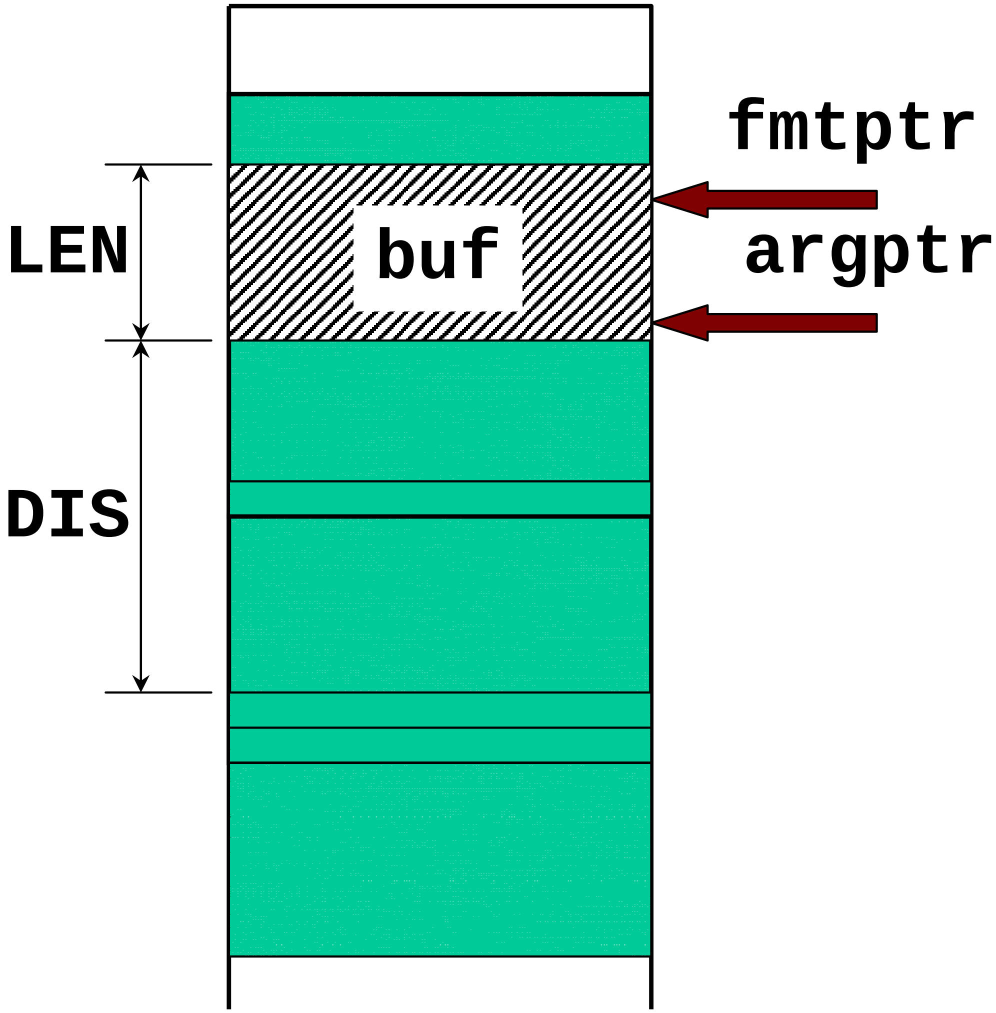


buf = “🟥”

Format-string exploits

What if we move **argptr** into **buf**?

Remember, attacker can control **buf**!



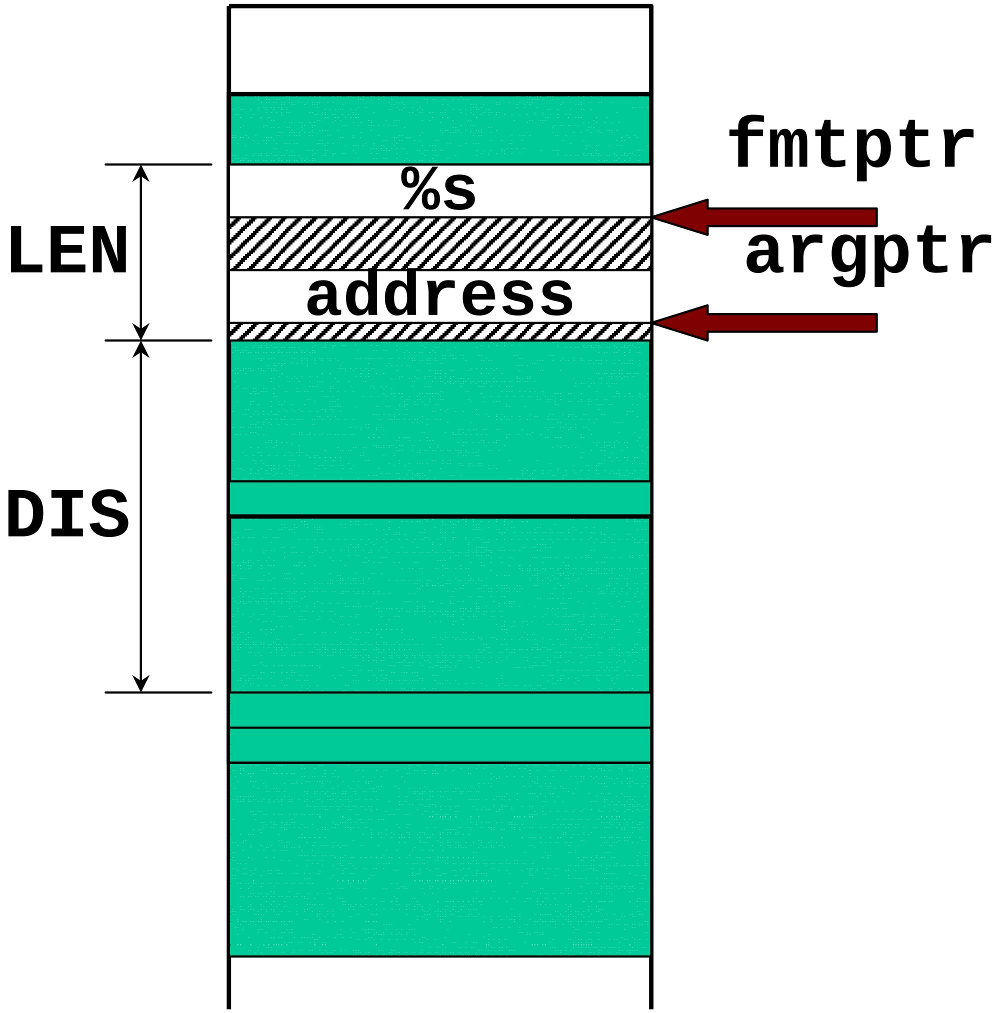
Format-string exploits

Example exploit scenario:

- **fmtptr** is at a “%s”
- **buf** contains an attacker-chosen address.
- **argptr** points to this location within **buf**

Can read from arbitrary memory location!

Can also write to arbitrary memory location, e.g. return addresses (paper has details)

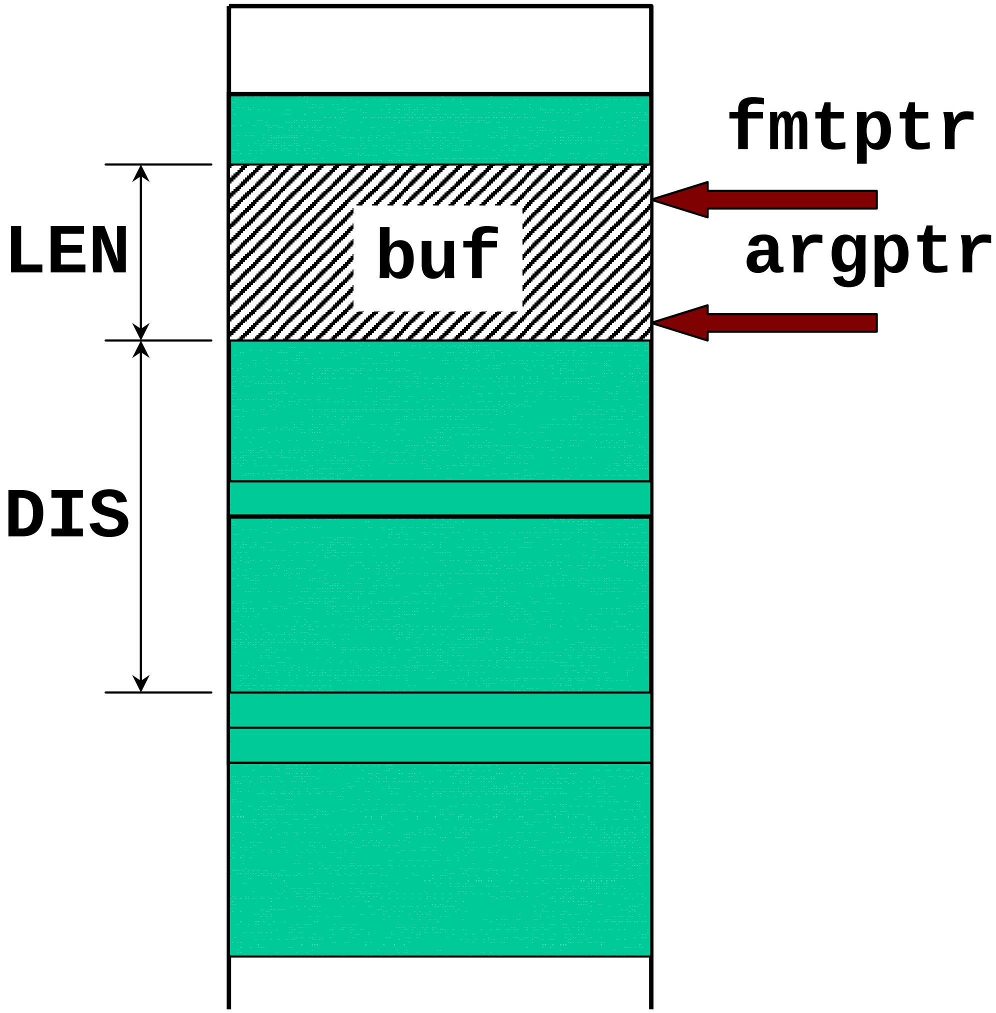


Format-string exploits

Exploit techniques well-known.

Key observations:

1. **DIS** and **LEN** completely characterize any **printf** call.
2. Each byte in **buf** instructs **printf** what to do next.



Format-string exploits

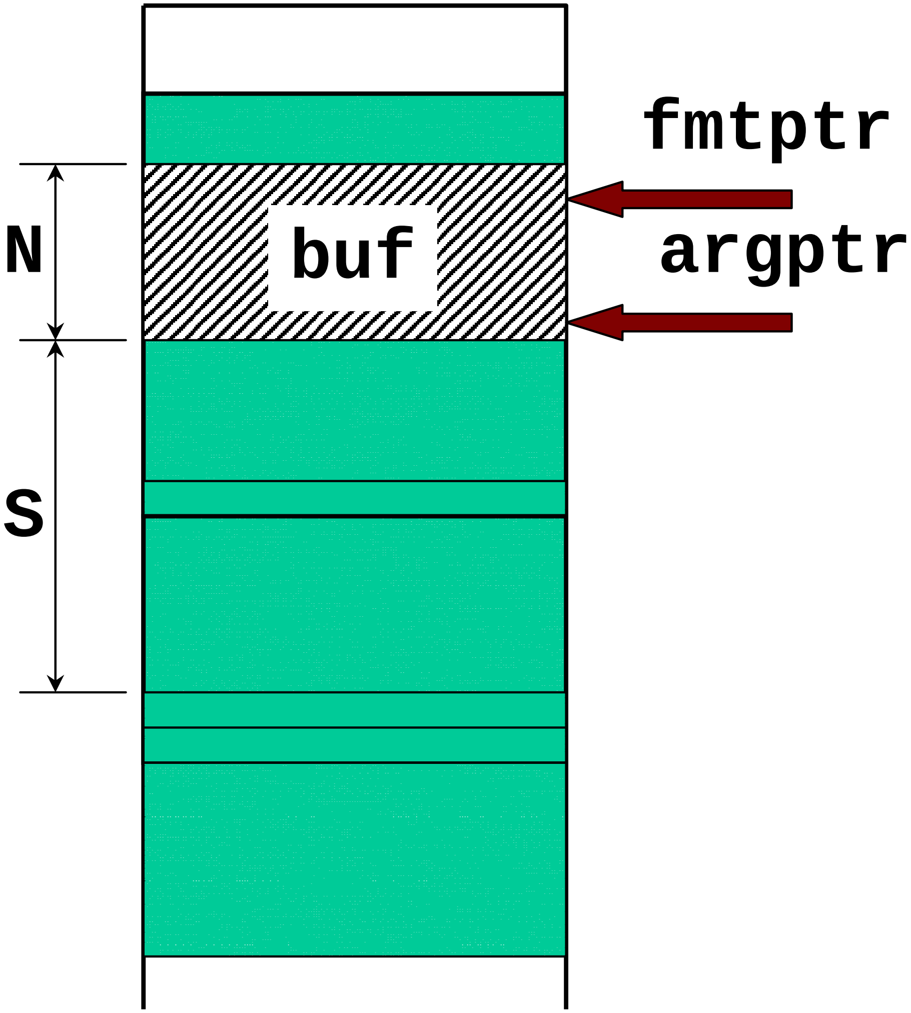
Format-string is a sequence of API operations.

Each byte of the format-string is an instruction.

Finding format-string exploits

=

Finding API-Level exploits



Finding format-string exploits

- Model how **`printf`** interprets format-string:
  - Encoded in the source code of **`printf`**.
  - Need to construct this model only once.
- S = (V,Init,Σ,L)
  - V: Various flags used in **`printf`** implementation that encode its state.
  - Σ: Set of all ASCII characters (size = 256).
  - L: All allowed format-strings (Σ^*)
    - Can restrict L to find exploits that follow a particular pattern

Finding format-string exploits

- For a vulnerable application:
  - Find **`DIS`** and **`LEN`**: How? Disassemble!
  - Formulate Bad.
  - Check against the model of **`printf`**.

- See paper for examples of Bad to:
  - Read from arbitrary memory location.
  - Write to an arbitary memory location.

Finding format-string exploits

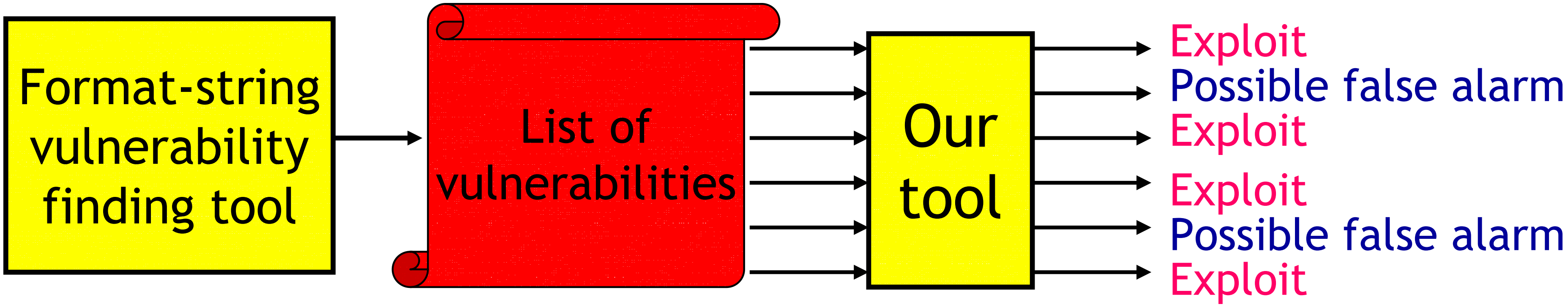
- The model of **`printf`**:
  - Requires precise reasoning about stack locations, in particular, the format-string.
  - Has integer operations: pointer arithmetic to advance **`fmtptr`** and **`argptr`**.
- Quantifier-free Presburger-arithmetic with theory of uninterpreted functions.
  - UCLID tool. [Bryant et al. CAV 2002]

Format-string exploit-detection tool

- Finds exploits against vulnerabilities in real-world software packages.
- Can find different kinds of exploits.
- Can find an arbitrary number of variations of a given exploit.
- Can work on binary executables.
- Can improve the quality of format-string vulnerability-detection tools.

Possible use scenario

- Percent-S [Shankar et al. USENIX Security 2001] finds possibly vulnerable locations. No exploits.
- Run our tool at each vulnerable location:
  - Exploit generated: true vulnerability.
  - No exploit generated: possibly a false alarm.



Results

- Exploits against vulnerabilities in real-world software:
  - See paper for details

Software	DIS	LEN	Exploit description
php-3.0.16	24	1024	Overwrite memory location
qpopper-2.53	2120	1024	Read a memory location
wu-ftpd-2.6.0	9364	4096	Overwrite memory location

Results

DIS	LEN	Read exploit	Write exploit
0	7	“$a_1a_2a_3a_4$%s”	No exploit
4	7	No exploit	No exploit
4	16	“$a_1a_2a_3a_4$%d%s”	“%234Lg%n$a_1a_2a_3a_4$”
4	16	“%Lx%ld%s$a_1a_2a_3a_4$”	“$a_1a_2a_3a_4$%%%229x%n”
8	16	“$a_1a_2a_3a_4$%Lx%s”	“$a_1a_2a_3a_4$%230g%n”
16	16	“%Lg%Lg%s$a_1a_2a_3a_4$”	“$a_1a_2a_3a_4$%137g%93g%n”
20	20	“$a_1a_2a_3a_4$%Lg%g%s”	“$a_1a_2a_3a_4$%210Lg%20g%n”
24	20	“$a_1a_2a_3a_4$%Lg%Lg%s”	“$a_1a_2a_3a_4$%61Lg%169Lg%n”
32	24	“$a_1a_2a_3a_4$%g%Lg%Lg%s”	“$a_1a_2a_3a_4$%78Lg%80g%72Lg%n”

Results

DIS	LEN	Read exploit	Write exploit
0	7	“$a_1a_2a_3a_4$%s”	🟥
4	7	🟥	🟥
4	16	“$a_1a_2a_3a_4$%d%s”	“%234Lg%n$a_1a_2a_3a_4$”
4	16	“%Lx%ld%s$a_1a_2a_3a_4$”	“$a_1a_2a_3a_4$%%%229x%n”
8	16	“a	$a_2a_3a_4$%230g%n”
16	16	“%Lg%Lg%s$a_1a_2a_3a_4$”	“$a_1a_2a_3a_4$%137g%93g%n”
20	20	“$a_1a_2a_3a_4$%Lg%g%s”	“$a_1a_2a_3a_4$%210Lg%20g%n”
24	20	“$a_1a_2a_3a_4$%Lg%Lg%s”	“$a_1a_2a_3a_4$%61Lg%169Lg%n”
32	24	“$a_1a_2a_3a_4$%g%Lg%Lg%s”	“$a_1a_2a_3a_4$%78Lg%80g%72Lg%n”

Ability to find false alarms

Results

DIS	LEN		
0	7	“$a_1a_2a_3a_4$%s”	No exploit
4	7	No exploit	No exploit
4	16	“$a_1a_2a_3a_4$%d%s”	“%234Lg%n$a_1a_2a_3a_4$”
4	16	“%Lx%ld%s$a_1a_2a_3a_4$”	“$a_1a_2a_3a_4$%%229x%n”
8	16		a_4%230g%n”
16	16		%137g%93g%n”
20	20		210Lg%20g%n”
24	20	“$a_1a_2a_3a_4$%Lg%Lg%s”	“$a_1a_2a_3a_4$%61Lg%169Lg%n”
32	24	“$a_1a_2a_3a_4$%g%Lg%Lg%s”	“$a_1a_2a_3a_4$%78Lg%80g%72Lg%n”

Ability to find different kinds of exploits: Parametrized by the predicate Bad

Results

DIS	LEN	Read exploit	Write exploit
0	7	“$a_1a_2a_3a_4$%s”	No exploit
4	7	No exploit	No exploit
8	16	“a_1	$_3a_4$%230g%n”
16	16	“%Lg	$_4$%137g%93g%n”
20	20	“$a_1a_2a_3a_4$%Lg%g%s”	“$a_1a_2a_3a_4$%210Lg%20g%n”
24	20	“$a_1a_2a_3a_4$%Lg%Lg%s”	“$a_1a_2a_3a_4$%61Lg%169Lg%n”
32	24	“$a_1a_2a_3a_4$%g%Lg%Lg%s”	“$a_1a_2a_3a_4$%78Lg%80g%72Lg%n”

Ability to find variants of an exploit

Talk structure

- Motivation and Overview.
- Framework for finding API-level exploits.
- Example: format-string exploit-detector.
  - Overview of `printf` and format-string exploits.
  - Instantiating `printf` in our framework.
  - Results.
  - Comparison with other tools.
- Related work.
- Conclusions.

Related work

- Software Model Checking [Blast,SLAM,Magic,CBMC]
  - Counter-example guided abstraction refinement.
  - Exploits ≈ Concrete counter-examples.
- Test generation [Beyer et al. ICSE04,Boyapati et al. ISSTA02]
  - Exploits can be used as test cases.
- Ad-hoc techniques [Thuemmel 2001,Newsham 2000]
  - No soundness guarantees. Cannot find variants.

Summary of important ideas

- Exploit-finding requires modeling low-level details of the system.
- Exploit-finding can benefit vulnerability-finding tools.
- Demonstrated using API-level exploits.

Automatic Discovery of API-Level Exploits

Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha,
Thomas W. Reps and Randal E. Bryant

University of Wisconsin and Carnegie Mellon University