CMSC414 Computer and Network Security

Mitigating Memory Safety Vulnerabilities, SQL Injection

Yizheng Chen | University of Maryland
surrealyz.github.io

Feb 12, 2026

Credits: some slides were from Dave Levin

Announcements

- Project 1, **due in 7 days! Thursday, February 19**
- If you still haven’t set up gitlab, it is very late now!
- Project 2 released, due on February 26

Agenda

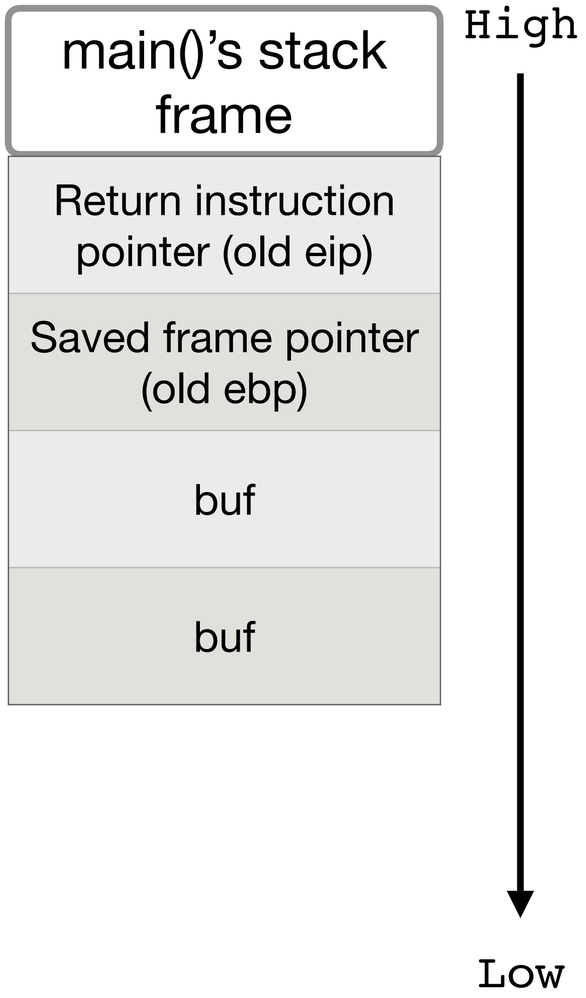
- Memory-safe languages
- Writing memory-safe code
- Building secure software
- Exploit mitigations
  - Non-executable pages
  - Stack canaries
  - Pointer authentication
  - Address space layout randomization (ASLR)
- Combining mitigations

Agenda

- SQL Injection
- Introduction to Web

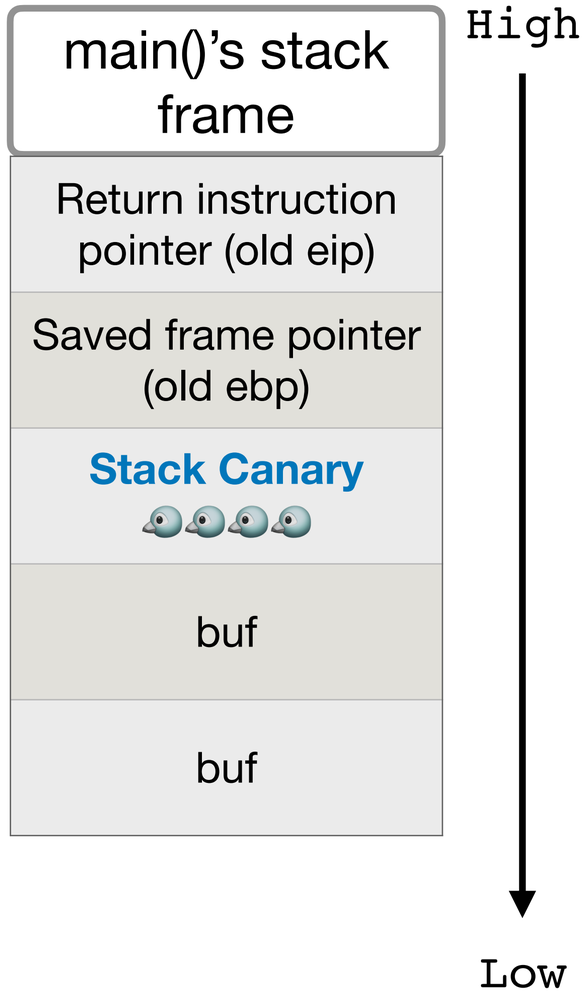
Regular Stack Example

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



Stack Canaries

```
void main() {
    vulnerable();
}

void vulnerable() {
    char buf[8];
    gets(buf)
    ...
}
```



The attack will have to overwrite the **stack canary**

Stack Canaries

- During runtime, generate a **random secret** value and save it in the canary storage
  - In the function prologue, place the canary value on the stack right below the SFP/RIP
  - In the function epilogue, check the value on the stack and compare it against the value in canary storage
  - If the canary value changes, somebody is probably attacking our system!

Stack Canaries

- A canary value is unique every time the program runs but the **same for all functions within a run**
- A canary value uses **a NULL byte as the first byte** to mitigate string-based attacks (since it terminates any string before it)
  - Example: A format string vulnerability with %s might try to print everything on the stack
  - The null byte in the canary will mitigate the damage by stopping the print earlier.
- Overhead: compiler inserts a few extra instructions, bust mostly low overhead

Subverting Stack Canaries

- **Leak** the value of the canary: Overwrite the canary with itself
- **Bypass** the value of the canary: Use a random write, not a sequential write
- **Guess** the value of the canary: Brute-force

Guess the Canary

- The first byte (8 bits) is always a NULL byte
- On 32-bit systems: 24 bits to guess
  - 32 - 8 = 24
  - 2^24 possibilities (~16 million), can be brute-forced, depending on the setting
  - 1 try/sec, 100 days
- On 64-bit systems: 56 bits to guess
  - 1,000 tries/sec, 2 million years

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
   - Mitigation: Stack Canaries
   - Mitigation: Pointer Authentication
4. Return from the function
5. Begin executing malicious shellcode

General idea:
put some secret values on the stack

- **Stack Canaries:** place some secret value below pointers (return instruction pointer and saved frame pointer)

- **Pointer Authentication:** place some secret value in the pointers

Pointer Authentication

- **Pointer Authentication:** place some secret value in the pointers
  - e.g., In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits

Pointer Authentication

- **Pointer Authentication:** place some secret value in the pointers
  - e.g., In a 64 bit system, 42 bits are ~4TB of memory, 22 bits are unused
  - Put the secret **(PAC, pointer authentication code)** in unused bits
  - Before using the pointer in memory, check if the PAC is still valid
    - Invalid: crash the program
    - Valid: restore unused bits, use the address normally

Example

- Address: 0x0000001234567899
- Suppose: 24 bits unused (zeros), 40 bits of addresses for a system
- PAC: 0xABCDEF
- Address on the stack: 0xABCDEF1234567899

Properties of PAC

- Each possible address has its own PAC
  - The PAC for the address 0x000000007ffffec0 is different from the PAC for 0x000000007ffffec4
- Message Authentication Code (MAC) in the cryptography lectures
- Only someone who knows the CPU's master secret can generate a PAC for an address
  - f(key, address)
- The CPU's master secret is not accessible to the program
  - Leaking program memory will not leak the master secret

Subverting Pointer Authentication

- Find a vulnerability to trick the program to generating a PAC for any address
- Learn the master secret
  - Vulnerability in the OS
- Guess a PAC: Brute-force (~ 2^20 possibilities)
- Pointer reuse
  - If the CPU already generated another PAC for another pointer, we can copy that pointer and use it elsewhere

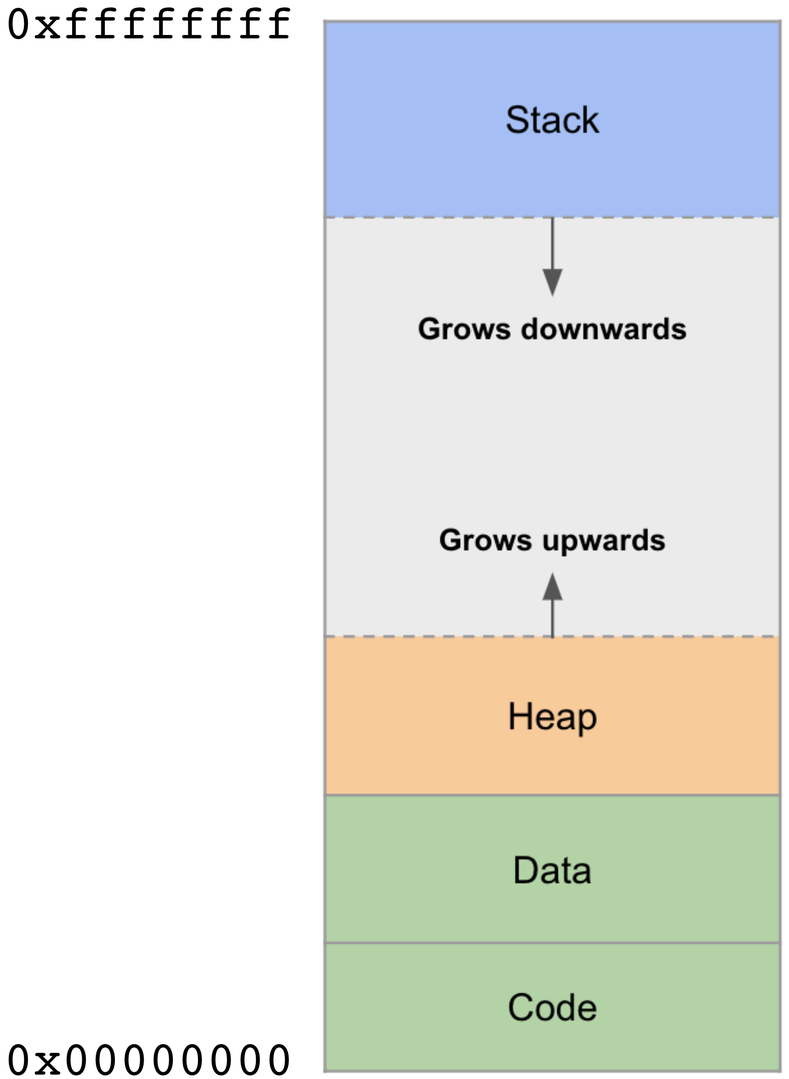
Recall: Putting Together an Attack

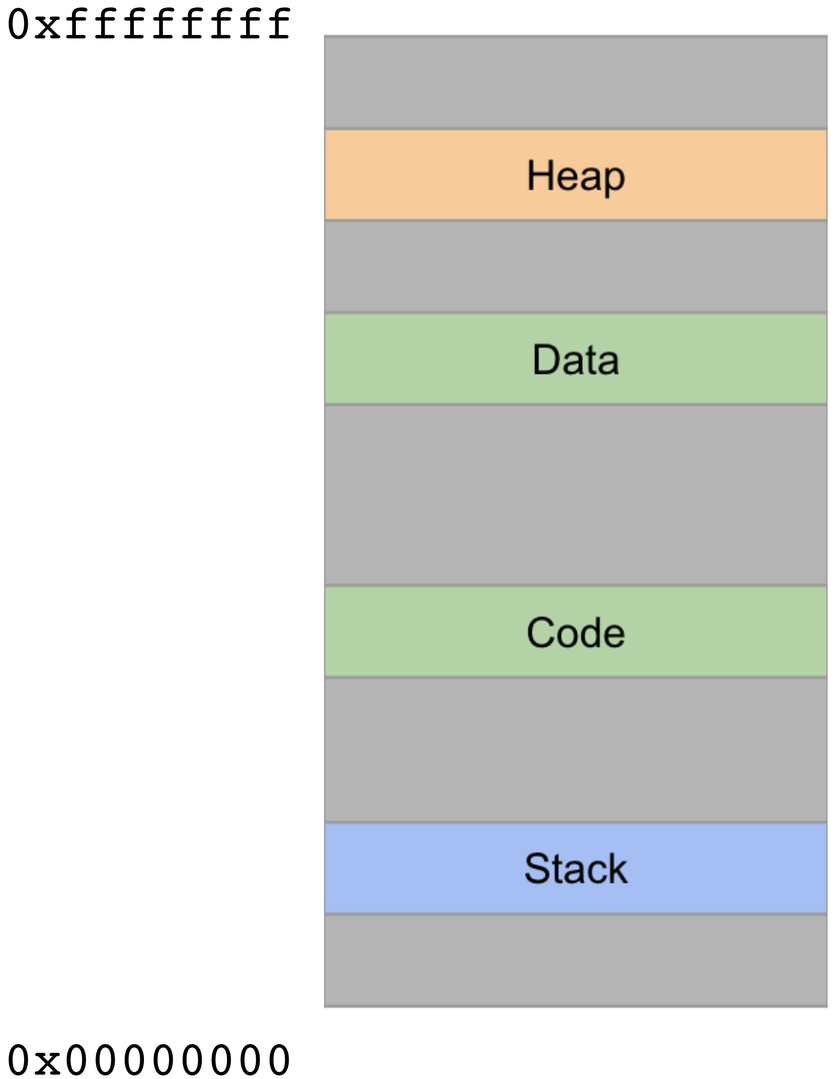
1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
   - Mitigation: Address Space Layout Randomization (ASLR)
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

Address Space Layout Randomization

- Goal: make it hard for attackers to place shell code on the stack, on the heap, or find out the address of the code
- Randomize the addresses of code, data, heap, stack
- Theoretically, very hard to know the addresses, so we can mitigate the attacks

Address Space Layout Randomization





Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead

Address Space Layout Randomization

- **Address space layout randomization (ASLR):** Put each segment of memory in a different location each time the program is run
  - Programs are dynamically linked at runtime, so ASLR has almost no overhead
- However…
- Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)
  - Leak the address of a pointer, whose address relative to your shellcode is known (stack pointer, RIP)
  - Guess the address of your shellcode: Brute-force

Combining Mitigations

- **Security Principle: Defense in depth**
  - Multiple types of defenses should be layered together so an attacker would have to breach all the defenses to successfully attack a system.
- Example: Combining ASLR and non-executable pages
  - To defeat these, the attacker needs to find two vulnerabilities
  - 1) find a way to leak memory and reveal the address randomization (defeat ASLR)
  - 2) find a way to write to memory and write a ROP chain (defeat non-executable pages)

Summary

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
   - Mitigation: Address Space Layout Randomization (ASLR)
3. Overwrite the RIP with the address of the shellcode
   - Mitigation: Stack Canaries
   - Mitigation: Pointer Authentication
4. Return from the function
5. Begin executing malicious shellcode
   - Mitigation: Non-executable pages

Agenda

- SQL Injection
- Introduction to Web

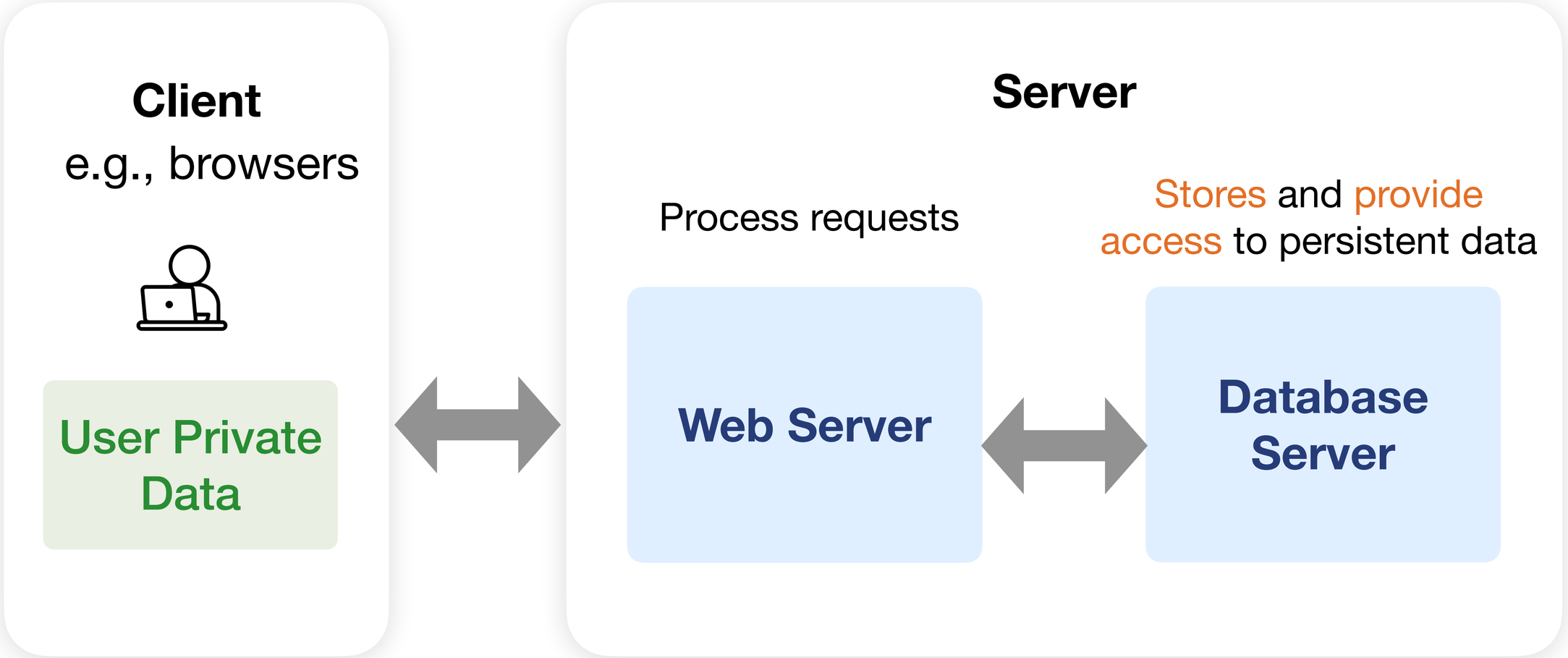
2025 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: X | View in table format | Key Insights | Methodology

1. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 7 | Rank Last Year: 1
2. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 4 | Rank Last Year: 3 (up 1) ▲
3. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 4 (up 1) ▲
4. Missing Authorization
   CWE-862 | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲
5. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 12 | Rank Last Year: 2 (down 3) ▼
6. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 10 | Rank Last Year: 5 (down 1) ▼
7. Use After Free
   CWE-416 | CVEs in KEV: 14 | Rank Last Year: 8 (up 1) ▲
8. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 3 | Rank Last Year: 6 (down 2) ▼
9. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 20 | Rank Last Year: 7 (down 2) ▼
10. Improper Control of Generation of Code ('Code Injection')
    CWE-94 | CVEs in KEV: 7 | Rank Last Year: 11 (up 1) ▲

https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

A Very Basic Web Architecture



Databases

- For this class, we will cover SQL databases:
  - SQL: Structured Query Language, to create and query data
  - Each database has some tables
  - Each table has a predefined structure, so it has columns for each field and rows for each entry
- Database server manages access and storage of these databases

SQL

- **Structured Query Language (SQL)**: The language used to interact with and manage data stored in a database
- Good SQL servers are **ACID (atomicity, consistency, isolation, and durability)**
  - Essentially ensures that the database will never store a partial operation, return an invalid state, or be vulnerable to race conditions
- Declarative programming language, rather than imperative
  - Declarative: Use code to define the result you want
  - Imperative: Use code to define exactly what to do (e.g. C, Python, Go)

Database Transactions

"Give me everyone in the User table who is listed as taking CMSC414 in the Classes table" 2 reads

"Deduct $100 from Alice; Add $100 to Bob" 2 writes

1 transaction

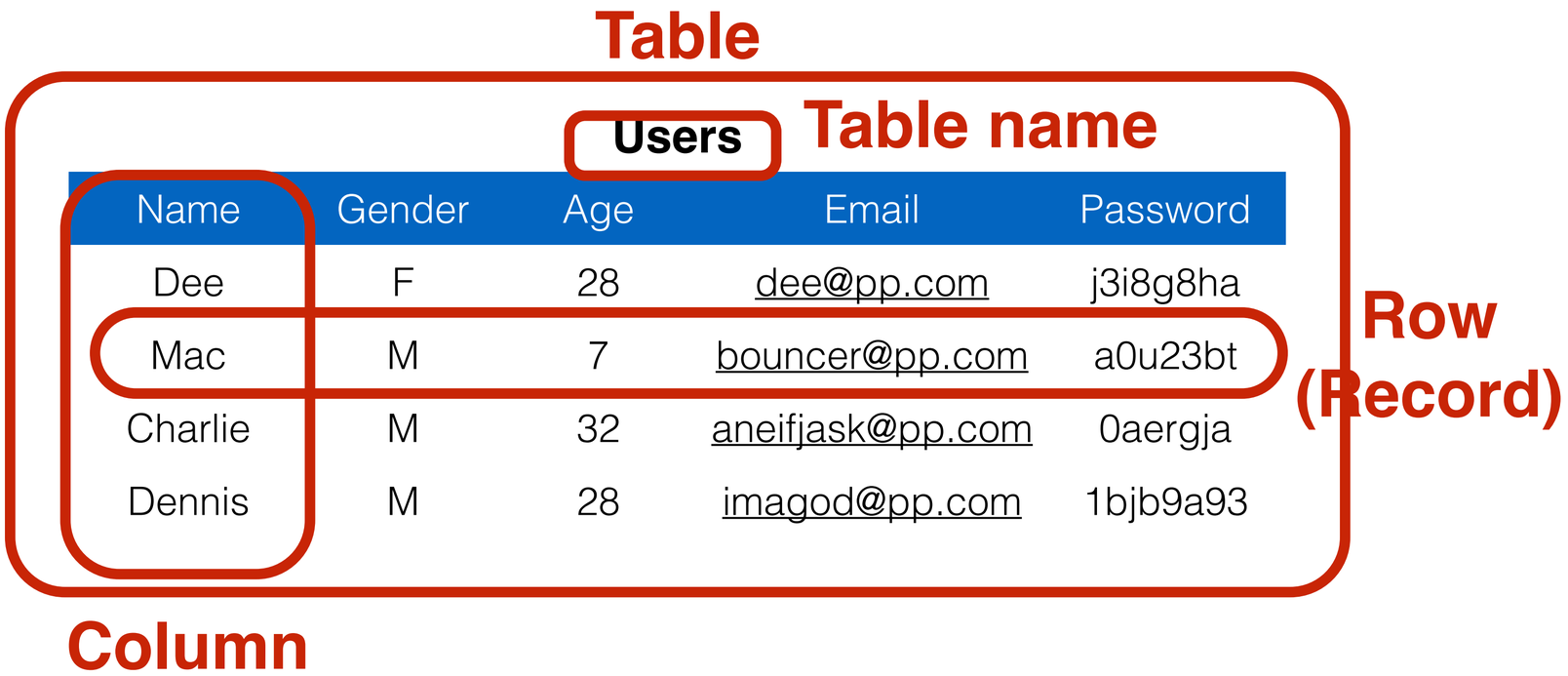
- A transaction is a unit of work in a database
- Good database servers are **ACID**
  - **A**tomicity: Transactions complete entirely or not at all
  - **C**onsistency: The database is always in a *valid* state (but not necessarily *correct*)
  - **I**solation: Results from a transaction aren't visible until it is complete
  - **D**urability: Once a transaction is committed, it remains, despite, e.g., power failures

TOCTTOU Vulnerability

- **Time-of-check to time-of-use** vulnerability
  - Check: no problem
  - Use: has problem

- Race condition
  - e.g., Reading in a state where other writes are in progress
  - e.g., Writing some partial content before finishing, and then another transaction reads

SQL Database Example



SQL (Standard Query Language) Example

Users

Name	Gender	Age	Email	Password
Dee	F	28	dee@pp.com	j3i8g8ha
Mac	M	7	bouncer@pp.com	a0u23bt
Charlie	M	32	readgood@pp.com	0aergja
Dennis	M	28	imagod@pp.com	1bjb9a93

```
SELECT Age FROM Users WHERE Name='Dee';
```
28

```
SELECT Age FROM Users WHERE Name='Dee' OR Name='Mac';
```
28, 7

```
UPDATE Users SET email='readgood@pp.com'
   WHERE Age=32; -- this is a comment
INSERT INTO Users Values('Frank', 'M', 57, ...);
DROP TABLE Users;
```

Some SQL Syntax

- SELECT * FROM table
  - The asterisk (*) is shorthand for “all columns.” Select all columns from the table, keeping all rows.
- WHERE can be used to filter out certain rows
  - Arithmetic comparison: <, <=, >, >=, =, <>
  - Arithmetic operators: +, - , * , /
  - Boolean operators: AND, OR
  - AND has precedence over OR

Server-side code

Website

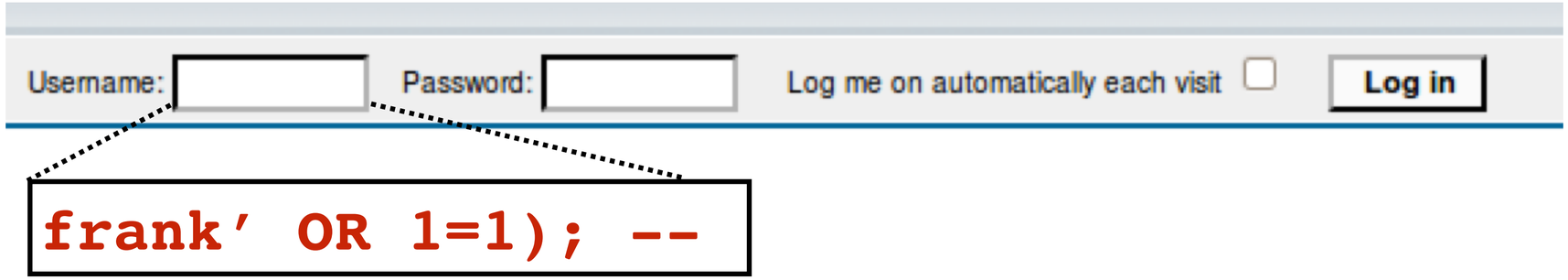
“Login code” (php)

```
$result = mysql_query(“select * from Users
        where(name=‘$user’ and password=‘$pass’);”);
```

Suppose you successfully log in as $user
if this query returns any rows whatsoever

How could you exploit this?

SQL injection




```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

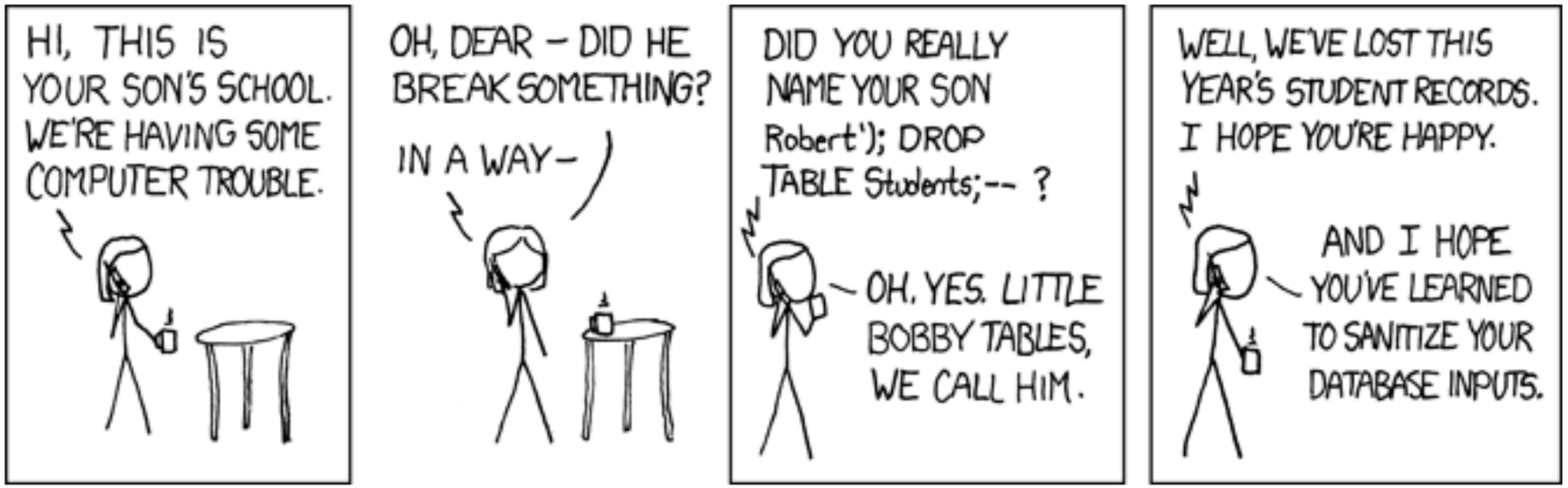
```
$result = mysql_query("select * from Users where
(name='frank' OR 1=1); -- ' and password='x');");
```

SQL injection


```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");

$result = mysql_query("select * from Users
        where(name='frank' OR 1=1);
        DROP TABLE Users; --
         ' and password='garbage');");
```

Can chain together statements with semicolon:
STATEMENT 1 ; STATEMENT 2

Exploits of a Mom



https://www.explainxkcd.com/wiki/index.php/327:_Exploits_of_a_Mom

A “Licence plate” with an SQL injection attack as a way to fight back traffic cameras.
https://www.reddit.com/r/geek/comments/1j9tn3/speed_camera_sql_injection/

SQL Injection Defense: Input Sanitization

- Blocklist: block special characters: ’ -- ;
  - But we want these characters sometimes! e.g., “Peter O’Connor”
- Allowlist: input within range, e.g., integer values for some fields
- Escape Inputs
  - **Escaper** adds \ in front of special characters: \; \’
  - **SQL parser** treats escaped char as literal char
- Building a good escaper can be tricky
  - Attacker uses \’, how does escaper and SQL parser work?
  - Secure escaper exists in SQL libraries
- Escaper may not be an effective solution, if we run SQL queries with raw user inputs

The Underlying Issue

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

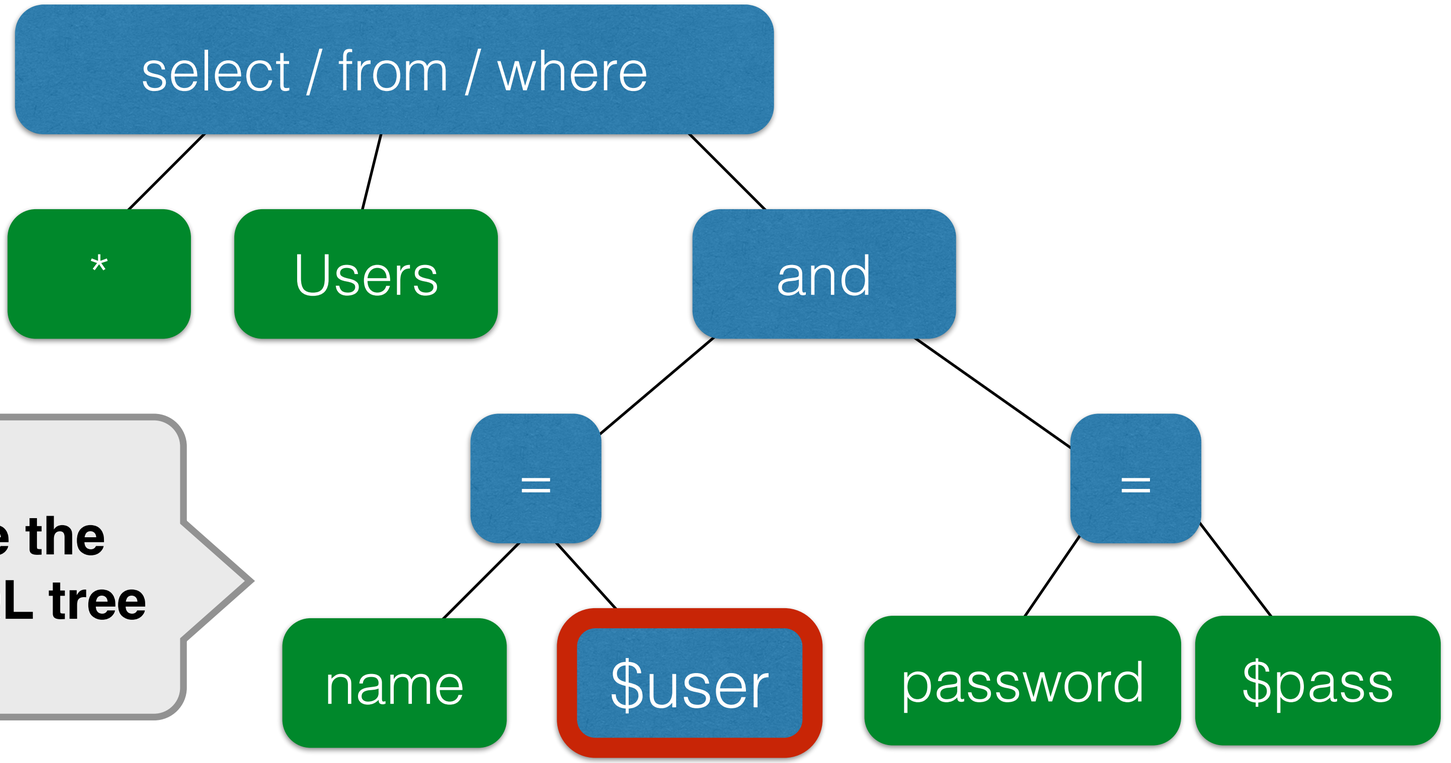
- This one string combines **code** and **data**
  - User input => SQL queries
- Similar to buffer overflow:
  - When the boundary between code and data blurs, we open ourselves up to vulnerabilities

Parameterized SQL / Prepared Statements

- **Idea: Parse the SQL query structure first, then insert the data**
- Use a question mark (?) for data when writing SQL statements
- When the parser encounters the ?, it fixes it as a single node in the syntax tree
- After parsing, only then, it inserts data
- **The untrusted input cannot change the SQL query structure**

Example without Prepared Statements

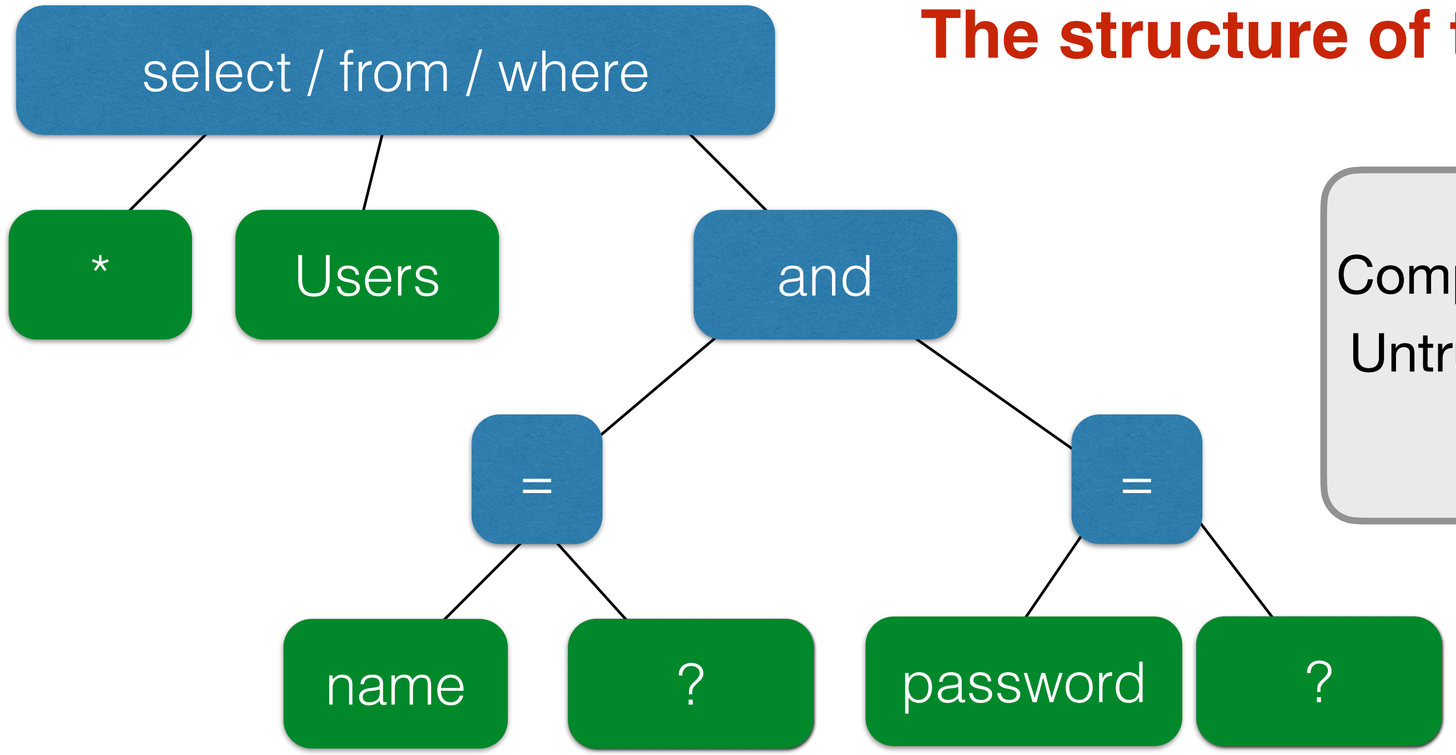
```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```



$user can change the structure of the SQL tree

Prepared Statement Example

```
$statement = $db->prepare("select * from Users
     where(name=? and password=?);");
```



The structure of the tree is *fixed*

Compile first, bind data later

Untrusted input will only be treated as data

Mitigate the Impact of Attacks

- The principle of least privilege
  - Limit commands and tables a user can access
  - e.g., Allow SELECT queries on Orders_Table but not on Creditcards_Table
- Encrypt sensitive data in the SQL table
  - May not need to encrypt Orders_Table
  - But certainly encrypt Creditcards_Table.cc_numbers

Followup Reading

Steve Friedl's Unixwiz.net Tech Tips

SQL Injection Attacks by Example

A customer asked that we check out his intranet site, which was used by the company's employees and customers. This was part of a larger security review, and though we'd not actually used SQL injection to penetrate a network before, we were pretty familiar with the general concepts. We were completely successful in this engagement, and wanted to recount the steps taken as an illustration.

Table of Contents

"SQL Injection" is subset of the an unverified/unsanitized user input vulnerability ("buffer overflows" are a different subset), and the idea is to convince the application to run SQL code that was not intended. If the application is creating SQL strings naively on the fly and then running them, it's straightforward to create some real surprises.

We'll note that this was a somewhat winding road with more than one wrong turn, and others with more experience will certainly have different -- and better -- approaches. But the fact that we were successful does suggest that we were not entirely misguided.

There have been other papers on SQL injection, including some that are much more detailed, but this one shows the rationale of **discovery** as much as the process of **exploitation**.

The Target Intranet

This appeared to be an entirely custom application, and we had no prior knowledge of the application nor access to the source code: this was a "blind" attack. A bit of poking showed that this server ran Microsoft's IIS 6 along with ASP.NET, and this suggested that the database was Microsoft's SQL server: we believe that these techniques can apply to nearly any web application backed by any SQL server.

The login page had a traditional username-and-password form, but also an email-me-my-password link; the latter proved to be the downfall of the whole system.

When entering an email address, the system presumably looked in the user database for that email address, and mailed something to that address. Since **my** email address is not found, it wasn't going to send **me** anything.

So the first test in any SQL-ish form is to enter a single quote as part of the data: the intention is to see if they construct an SQL string literally without sanitizing. When submitting the form with a quote in the email address, we get a 500 error (server failure), and this suggests that the "broken" input is actually being parsed literally. Bingo.

We speculate that the underlying SQL code looks something like this:

```
SELECT fieldlist
  FROM table
 WHERE field = '$EMAIL';
```

Here, **$EMAIL** is the address submitted on the form by the user, and the larger query provides the quotation marks that set it off as a literal string. We don't know the specific *names* of the fields or table involved, but we do know their *nature*, and we'll make some good guesses later.

http://www.unixwiz.net/techtips/sql-injection.html

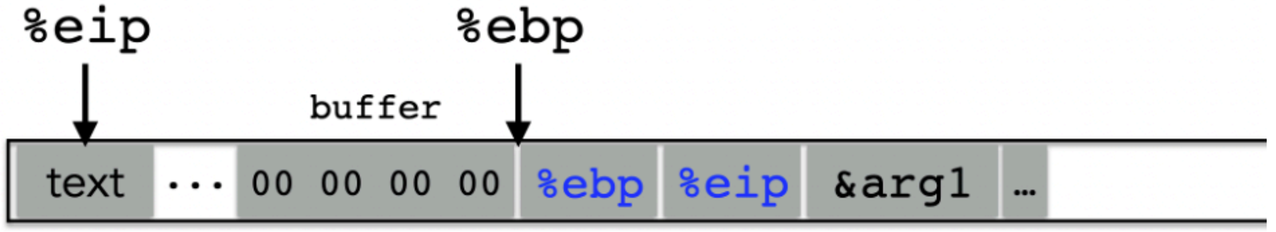
Tutorial on Computer

run <input>	Run the program with `input` as the command-line arguments
print <var> (or just "**p <var>**")	Print the value of variable `var` (Can also do some operations: `p &x`)
b <function>	Set a breakpoint at `function`
s **c**	**s**tep through execution (into calls) **c**ontinue execution (no more stepping)

`info frame` (or just "**`i f`**")	Show **i**nfo about the current **f**rame (prev. frame, locals/args, %ebp/%eip)
`info reg` (or just "**`i r`**")	Show **info** about **reg**isters (%ebp, %eip, %esp, etc.)
`x/<n> <addr>`	E**x**amine <n> bytes of memory starting at address <addr>

gdb example

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



Set a breakpoint at func()

Run the program

Breakpoint reached

Print buffers' addr

Frame info

Current/saved eip

Where on the stack registers are saved

```
Reading symbols from example.x...
(gdb) b func
Breakpoint 1 at 0x11d5: file example.c, line 5.
(gdb) run
Starting program: /home/cmsc414/example.x
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (arg1=0x56557008 "AuthMe!") at example.c:5
5       }
(gdb) p &buffer
$1 = (char (*)[4]) 0xffffd4d8
(gdb) info frame
Stack level 0, frame at 0xffffd4f0:
 eip = 0x565561d5 in func (example.c:5); saved eip = 0x56556242
 called by frame at 0xffffd520
 source language c.
 Arglist at 0xffffd4e8, args: arg1=0x56557008 "AuthMe!"
 Locals at 0xffffd4e8, Previous frame's sp is 0xffffd4f0
 Saved registers:
  ebx at 0xffffd4e4, ebp at 0xffffd4e8, eip at 0xffffd4ec
```