

Data security lecture 7

21st October final

Teach via zoom rest of lectures – next week

Ex 7 – try to solve it and Ex 6 might be optional

Buffer overflow

Stack smashing last week

Today: race conditions

Integer overflow

Input validation, sql injection

Race conditions

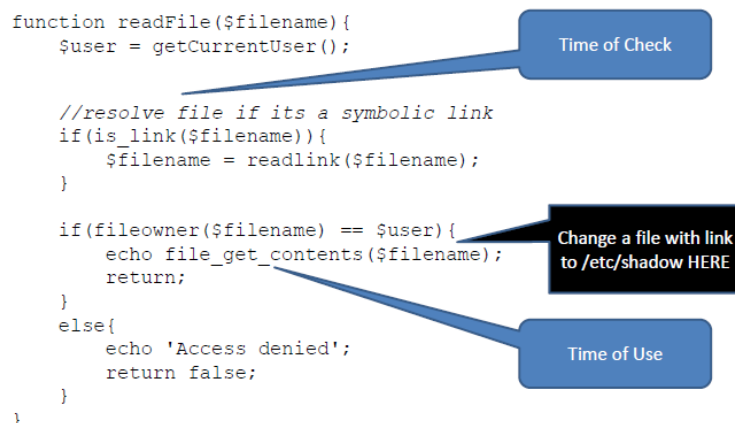
- Arise when security-critical process occurs in stages
 - Attacker makes change between stages
 - Between authorization and use
 - For example – replacing the software update file after its signature is verified (iOS jailbreak)
 - TOCTOU – time of check, time of use
 - Security processes should be atomic
- Diversion or alternative ways to get inside the system

Or get inside with nothing, no weapons

- Inside the mall, get your weapons

Race conditions – security processes happen in stages , that's when we attack

TOCTOU example



Php code

Between time of check and time of use, the attacker can change the file contents

Symbolic link – shortcut

File that points to another file... execute the file that it points to, not itself

After verified user owner of file, etc... then attacker changes the link and the system will run the corrupted or malicious file

Linux has the shadow password file, so can drop the file there to read it outside

Can this work in other languages?

- Yes, but it depends on the OS not the code itself (as the OS is responsible for the files)
 - o How to protect

How does the OS protect against it...

- Attacker needs to examine the code if its OSS
 - o But in windows, harder to see – attackers released the leaked code
 - o Reverse engineering to analyze binary code
- Block option of changing things after processes start to run/be executed
 - o Don't allow interference, or changes

- Integer overflow

- •One of the most popular exploits
- •Usually prepares grounds for memory exploits
- •Programmers rarely think about rare overflow cases..
- •Not only C/C++ -see [exploits of overflow in Java](#)

Common attack and exploits

Integer overflow explained

- Standard unsigned integer types

Type	Bits	Range	x86-32
unsigned char	8	0 - 255	255
unsigned short int	16	0 - 65,535	65,535
unsigned int	16*	0 - 65,535	4,294,967,295
unsigned long int	32	0-4,294,967,295	4,294,967,295
unsigned long long int	64	0 – $2^{64} - 1$	0 – $2^{64} - 1$

*at least

- What is the result of:

```
unsigned char c = 255 + 1
```

Data types and languages

32 bit OS range for char is different

1 byte = 8 bits

Max for 32 bit machine is 4 bytes

255 + 1 could go to 0 – wrap around but depends on compiler

- Cycle

Neg number = large for unsigned

Integer overflow explained

That's how its executed

250 + 8 = 2

256 numbers (count form zero so include it)

For loop runs forever

- Always ≥ 0
- # less than zero = large #

Changed iteration var many times

- Unsigned integers **wraparound (modulo MAX+1)**
- E.g., unsigned char:
the result of 250+8 is 258 modulo 256, which is 2
- What is wrong with the following example:

```
for (unsigned int i = n; i >= 0; i--)
```

- Real-life example:
 - 1100 flights were grounded due to a crash of a flight-crew-scheduling software ([Comair, 2004](#))
 - The SW used a 16-bit counter, limiting the number of changes to 32,768 a month
 - Storms -> too many changes -> system crash



Integer overflow exploitation

```
int* myfunction(int *array, unsigned int len)
{
    int *myarray; unsigned int i;
    myarray = malloc(len * sizeof(int));
    for(i = 0; i < len; i++)
        myarray[i] = array[i];
    return myarray;
}
```

- If len is large, len * sizeof(int) will wraparound
- The consequent copy will overwrite the heap!

Using C, C++

Malloc dynamic allocation in heap

No length check – negative numbers

Malloc multiplies

Size of int = 4 bytes

Huge # bigger than max value / 4 will wrap around and allocates very small amount of memory

- And exploit is my array is small but actual array is big and will get random memory in the array (same as last time) – with buffer overflow

Alloc small amount mem but give huge array and program will put large array in small # of memory

Compiler will then use more memory and run over other areas of memory

- Exploits integer overflow (bug in function w/ c language)

This is a heap overflow (not buffer overflow) b/c malloc

Most likely will crash but we can insert other code to execute

Integer overflow explained

- Standard signed integer types

Type	Bits	Range	x86-32
char	8	-128- to 127	
short int	16	-32,768 to 32,767	
int	16*	-32,768 to 32,767	-2,147,483,648 to 2,147,483,647
long int	32	-2,147,483,648 to 2,147,483,647	-2,147,483,648 to 2,147,483,647
long long int	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Instead of using range from 0 to 255, we can split the range, half negative and positive #s

Same range , 256 options

Asymmetric because 0 is counted from -128 to 127

Int range gets moved

Signed Integer Type - Sign and Magnitude

- The sign bit represents whether the value is negative (sign bit set to 1) or positive (sign bit set to 0)
- The other value bits represent the magnitude of the value in pure binary notation
- For Example:

00101011 = 43

10101011 = -43

bit representation

MSB is signed indicator

Signed Integer Type - Two's Complement

- Result of subtracting the number from 2^N
- To negate a two's complement value:
 - Toggle each bit, including the sign bit
 - Add 1 (with carries as required)
- For Example:
00101011 = 43
11010101 = -43

other way

Sum the two gets zero

Ignore the carry otherwise it will be 1 final carry

8 bits to 9 bits (not saved as out of range)

Overflowing signed integers

According to C/C++ standard the signed overflow behavior is undefined
Compiler can do what ever it wants!

- Breaking the program silently in unpredictable ways, depending on optimization
- Changing the behavior with compiler updates

In practice overflowing will usually result in negative values

Value	Representation
127	01111111
...
1	00000001
0	00000000
-1	11111111
...
-127	10000001
-128	10000000

Add 1 to highest #

127 + 1 then get -128

But wrap around to other side (mirror)

- In negative #s
- -127 (add 1 at end)

This behavior is compiler dependent, we don't know what happens

Compiler tries to optimize things b/c of that it can decide on certain decisions which aren't predictable

Overflow usually with - #s

Signed overflow exploitation

```
int get_two_vars(int sock, char *out, int len)
{
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;
    if(recv(sock, buf1, sizeof(buf1), 0) < 0)
        { return -1; }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0)
        { return -1; }
    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));
    size = size1 + size2; /* [1] */
    if(size > len)
        { /* [2] */ return -1; }
    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);
    return size;
}
```

C code

Socket = shared memory

Pointer to buffer and its length

Recv reads from socket to buffer

Then memcpy – read from location to memory

Then sum them up

Now check if the size > length – if so don't continue

If less we now use memcpy to copy the buffer1 to out and add size1 to buffer

size = size1 + size2; /* [1] */ can be negative value, less than len and then continue to copy to memory a huge buffer size

```
if(recv(sock, buf1, sizeof(buf1), 0) < 0)
    { return -1; }
if(recv(sock, buf2, sizeof(buf2), 0) < 0)
    { return -1; }
/* packet begins with length info */
memcpy(&size1, buf1, sizeof(int));
memcpy(&size2, buf2, sizeof(int));
size = size1 + size2; /* [1] */
if(size > len)
    { /* [2] */ return -1; }
memcpy(out, buf1, size1);
memcpy(out + size1, buf2, size2);
return size;
```

size1 = 0x7fffffff
size2 = 0x7fffffff
size1 + size2 = -2!

Arbitrary memory write!

Len always > 0

Signed integer overflow could be exploited

Problematic conversions

- Mixing signed and unsigned – BAD idea (read [here](#) and [here](#))
 - Rules for conversion are complex
 - Converted to signed if fits, otherwise to unsigned
 - Constants are always signed

```
int copy_something(char *buf, int len)
{
    char kbuf[800];
    if(len <= 800)
        return memcpy(kbuf, buf, len);
}
```

```
int main (void)
{
    long a = -1;
    unsigned b = 1;
    printf ("%d\n", a
> b);
    return 0;
}
```

Prints 0 in 64 bits
Prints 1 in 32 bits

- Truncating
unsigned int ui = 300;
unsigned char uc = ui;
uc = 300 – 256 = 44

memcpy expects
unsigned int
Passing len = -2 will cause
buffer overwrite

64 bit machine will work fine with code – comparing two types that aren't the same

Unsigned and long are not the same

And if we try to compare then, convert the unsigned #

64 bit machine can convert unsigned #

4 bill unsigned int vs long – 2 billion

Turn sign to unsigned and get a huge number

A > B = 1 always will print 1

1 = true, 0 = false

Conversion type conventions are complex and each machine is different

Constants always signed

-2 input gives overflow

Memcpy expects unsigned # - we give it – num

Then get a huge number for allocation

Truncating

- Happens with different type assignments
- Subtracts highest value in range we want to assign
- If still too big, it keeps reducing via same method till it can get down to size it can hold

Short = half unsigned int

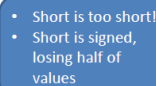
Strln = unsigned int

Overflow detection and mitigation

- Proper type selection (large enough, matches the operated types)

```
short total = strlen(argv[1]) + 1;  
size_t total = strlen(argv[1]) + 1;
```

- Range checking
 - Mostly forgotten
 - Causes code bloat
- Compiler features (GCC and clang -ftrapv flag)
- [Automatic detection tools](#)



- Short is too short!
- Short is signed, losing half of values

Use proper types on both sides to fix it

How to avoid these issues

Hard to remember all the types etc

Can't also cover each line in code with checks will bloat the code

More code = more confusion

Pro coders can also use overflows intentionally for optimization so it's a tradeoff

Overflow detection and mitigation

- The [CERT C Secure Coding Standard](#) has several rules to prevent range errors:

–INT30-C. Ensure that unsigned integer operation do not wrap

–INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data

–INT32-C. Ensure that operations on signed integer do not result in overflow

Range checks are not trivial

- Is this check enough?

```
unsigned int i, sum;
//set values to i and sum...
if (sum + i < UINT_MAX)
    sum += i
```

- No. It can wraparound too...
- The fix:

```
if (i <
    UINT_MAX - sum)
```



Fiora Aeterna
@FioraAeterna



Follow

security advice: always used signed integers. signing prevents data from being modified undetected by an adversary

Does this protect from int overflow

Sum and I can still be larger than max, so wrap around can still occur

Otherwise

The fix is

If (i < UINT_MAX - sum)

Make sure sum isn't greater than the OS max

If sum > max, then don't continue the loop, just stop

10

Input validation

- Range checks are an example of input validation

Consider: strcpy(buffer, argv[1])

A buffer overflow occurs if

len(buffer) < len(argv[1])

Software must **validate the input** by checking the length of argv[1]

- Becomes hard if the language is complex

Check range of values we get

Server side validation

- Consider web form data
- Suppose input is validated on client

For example, the following is valid

<http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=205>

- Suppose input is not checked on server
- Why bother since input checked on client?
- Then attacker could send http message

<http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=25>

User buying item, can change the arguments as they see fit

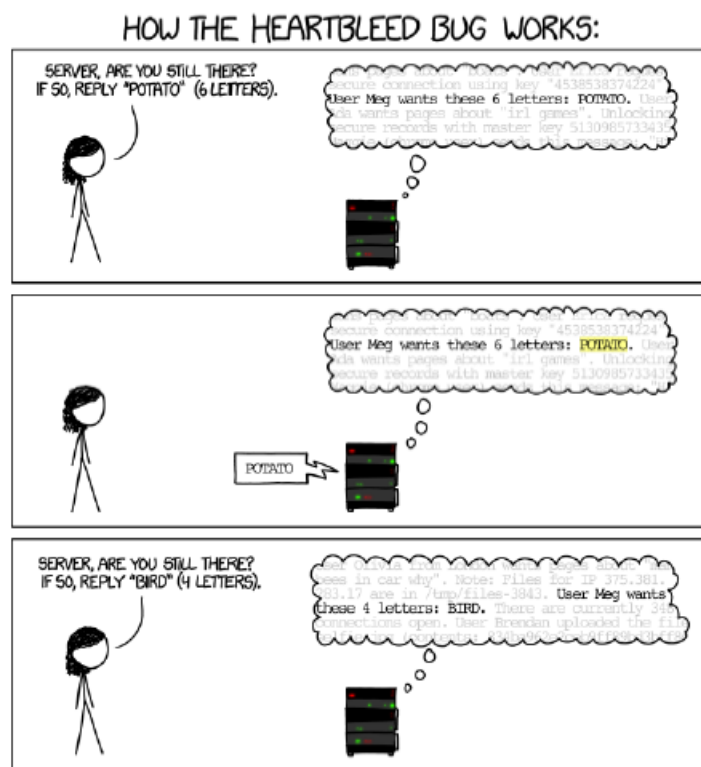
Make it cheaper or more expensive

Where do we validate the user input?

We don't trust the user machine – everything we get from them is untrusted

We need to check it all on **server**

Input validation - Heartbleed



Command injection

- Don't trust the user to do what he was supposed to!

```
filename = GetFileName()  
system ("ls -l " + filename + "> output")  
print output;
```

```
➤ Enter file name  
➤ file; cat /etc/shadow    Prints the passwords file
```

```
filename = GetFileName()  
filename = CanonicalFileName(filename)  
system ("ls -l " + filename + "> output")  
print output;
```

Get something that should have been protected by the system

Solution would be

Canonical form – remove all the semi colons and slashes etc

- Just leaves us with a string, should be a valid input

SQL injection

-

Similar to command

injection

-

Uses SQL to abuse

database commands

-

Exploits the lack of input

validation or

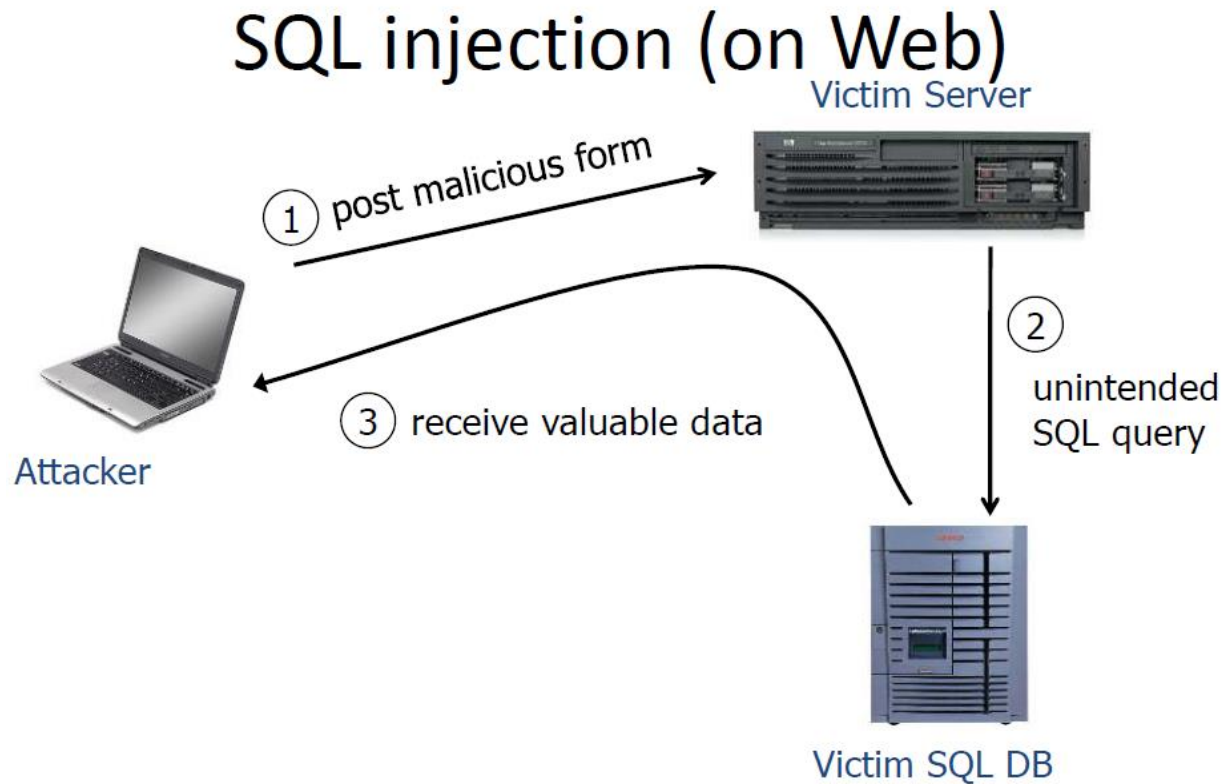
canonicalization

-

One of the top web site

Vulnerabilities

SQL – language to handle databases



Attacker expects the get stuff from DB

Sample Query

```
SELECT <columns> from <tbl> where <exp>
```

```
select * from comments  
where user_id = 2;
```



```
2, 2, "I like sugar"  
2, 3, "But not milk"
```

user_id	comment_id	comment
1	1	Test Comment
2	2	I like sugar
2	3	But not milk
3	4	Gordon is silly

comments

Command form

* wildcard – everything

Prints everything from user

Tautologies

```
SELECT <columns> from <tbl> where <exp>
```

```
select * from comments  
where user_id = 2  
OR 1= 1;
```



```
1, 1, "Test Comment"  
2, 2, "I like sugar"  
2, 3, "But not milk"  
3, 4, "Gordon is silly"
```

user_id	comment_id	comment
1	1	Test Comment
2	2	I like sugar
2	3	But not milk
3	4	Gordon is silly

comments

Tautologies are
useful for attacks

attacker now exploits system via

tautologies

Always true statement added to request

Database queries with PHP

- Sample PHP

```
$recipient = $_POST['recipient'];  
$sql = "SELECT PersonID FROM Person  
WHERE Username=".$recipient;  
$rs = $db->executeQuery($sql);
```

- What if 'recipient' is a malicious string that changes the meaning of the query?

```
$sql = "SELECT PersonID FROM Person  
WHERE Username=x or 1=1";
```

Gets all
IDs!

Php example

Add to the condition

Another example to show how widespread this attack is

Try to protect via quotation mark and apostrophies

Input = string

Add command after

Or – to ignore rest of string

To ignore the password part

DROP table

- Remove table
- Etc

SQL injection is a huge issue

More SQL injection attacks (ASP)

```
set ok = execute( "SELECT * FROM Users  
WHERE user=' " & form("user") & " '  
AND pwd=' " & form("pwd") & " ' " );
```

```
if not ok.EOF  
    login success  
else fail;
```

```
ok = execute( SELECT ...  
WHERE user= ' ' or 1=1 -- ... )
```

The "--" causes rest of line to be ignored.

```
ok = execute( SELECT ...  
WHERE user= ' ' ; DROP TABLE Users ... )  
attacker can add users, reset pwds, etc.
```

More SQL injection attacks

```
set ok = execute( "SELECT * FROM Users  
WHERE user=' " & form("user") & " '  
AND pwd=' " & form("pwd") & " ' " );
```

```
if not ok.EOF  
    login success  
else fail;
```

```
user =  
'; exec cmdshell  
'net user badguy badpwd'/ADD --
```

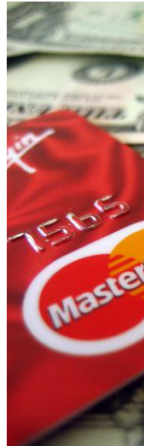
Then script does:

```
ok = execute( SELECT ...  
WHERE username= ' ' ; exec ... )
```

If SQL server context runs as "sa", attacker gets account on DB server

CardSystems attack

- CardSystems
 - credit card payment processing company
 - SQL injection attack in June 2005
 - put out of business
- The Attack
 - 263,000 credit cards stolen from database
 - credit cards stored unencrypted
 - 43 million credit cards exposed



Stole user credentials too

Isracard had fraud detection in place

What I have heard (from a trusted source) is that a SQL Injection vulnerability was exploited, the attacker created a Job in the database server that pulled out new records every 4 (?) days. This is a very easy attack since most database servers allow scheduling of actions as Jobs.

We have developed similar and new attacks that allows to steal complete databases from Internet, I hope we will be presenting this at next Black Hat :)

Cesar.

Command leaked info every 4 days

Didn't catch guys who did it



How to protect

- Parameters escaping: ' → \'
- Use pre-built SQL queries (ensures arguments are converted to proper types)



```
SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE
    username = @User AND
    password = @Pwd", dbConnection);

    cmd.Parameters.Add("@User",
Request["user"] ); ...

    cmd.ExecuteReader();
```

Odea is canonical form

Remove all commands

Another method is escaping

Use \ before each sign given as input

String instead of command

\n \t flags to convert things to trings = not command db not do nothing

Put another rbackslash to break this method