Secure Software Engineering

قسم علوم الحاسوب

فرع البرمجيات-المرحلة الرابعة

أ.د.شيماء حميد شاكر

Main Topics

- Fundamentals of software security.
- Software security design principles.
- Building secure software systems.
- Common software vulnerabilities.

What is Software Security?

- Software security is the idea of engineering software so that it continues to function correctly under malicious attack.
- Software Security aims to avoid security vulnerabilities by addressing security from the early stages of software development life cycle.
- "Security is a risk management."

Lecture one:

Introduction

Definition: software security

Software security—the idea of engineering software so that it continues to function correctly under malicious attack—is not really new, but it has received renewed interest
over the last several years as reactive network-based security approaches such as firewalls have proven to be ineffective. Unfortunately, today's software is riddled with
both design flaws and implementation bugs, resulting in unacceptable security risk

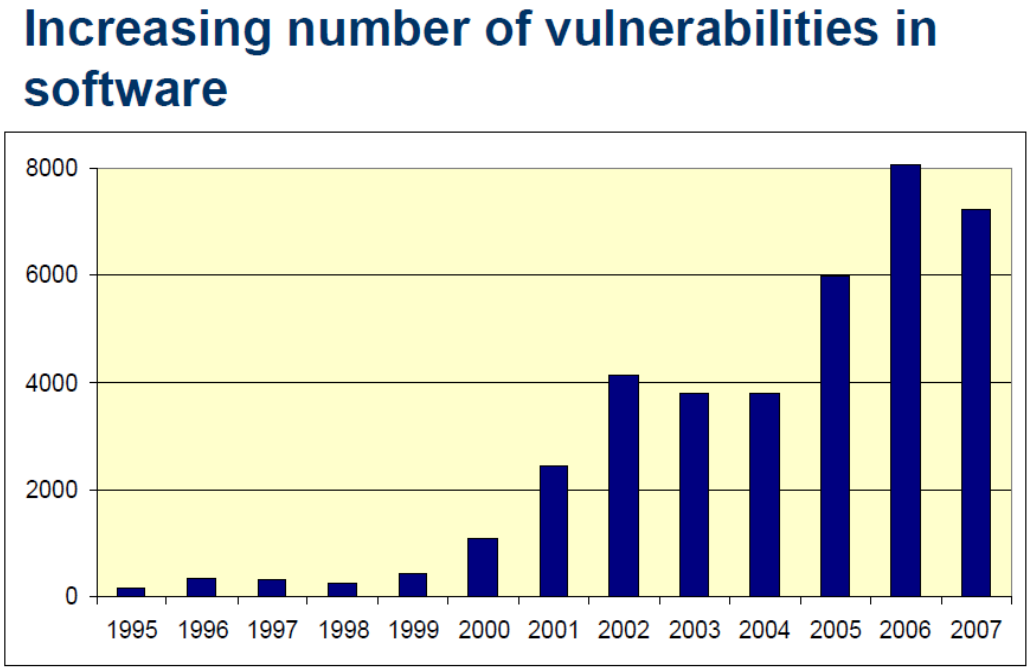
- The idea of engineering software so that it continues to function correctly under malicious attack.
- The process of designing, building, and testing software for security.
- Software Security aims to avoid security vulnerabilities [HW] by addressing security from the early stages of software development life cycle.
- Security is a risk [HW] management.

Why software security?

- Most software systems today contain numerous flaws and bugs that get exploited by attackers.
- New threats emerge everyday.
- Exponential increase in vulnerabilities in software systems.
- Programmers have a long history of repeating the same security-related mistakes!

The security problem

The security of computer systems and networks has become increasingly limited by the quality and security of the software running on constituent machines. Internet-enabled software, especially custom applications that use the Web, are a sadly common target for attack.



Vulnerability statistics
from CERT CC

Software is everywhere. It runs your car. It controls your cell phone. It keeps your dishwasher going. It is the lifeblood of your bank and the nation's power grid. And sometimes it even runs on your computer. What's important is realizing just how widespread software is. As businesses and society come to depend more heavily on software, we have to make it better. Now that software is networked by default, software
security is no longer a luxury—it's a necessity.

Why the problem is growing??

- **Connectivity:** through the Internet.
- **Extensibility**: updates or extensions(e.g. viewer in browsers)
- **Complexity:** growth in size and complexity of software systems
- H.W: why?

Security Problems in Software

- Software security, that is, the process of designing, building, and testing software for
- security, gets to the heart of computer security by identifying and expunging problems in
- the software itself. In this way, software security attempts to build software that can
- withstand attack proactively.

Recent stories:

- 2012 - A security flaw in Google Wallet that leads into full access to your Google Wallet account without extra app or rooting. Your Google Wallet account is tied to the device itself but not to the account.
- 2011 -Oracle's MySQL.com hacked via SQL Injection Attack!!
- 2010 -Hacker gained access to the Royal Navy website using SQL injection attack.
- 2009 -A security flaw in the Spotify service, by which private account information were exposed.
- 2006 -Hacker gained access to 800,000 UCLA students, faculty, and staff data.
- 2005 -A buffer overflow was deducted in Symantec pcAnywhere that could lead into a denial of service attack.

- 2005 -University of Southern California's online system applications were vulnerable to SQL
  Injection

Terminology

- **Defects**: are implementation vulnerabilities and design vulnerabilities.
- **Bug**: are implementation-level errors that can be detected and removed. E.g. buffer overflows
- **Flaws**: are problems at a deeper level. A design-level or architectural software defect. High-level defects cause 50% of software security problems
- **Failures**: are the inability of the software to perform its required function.
- **Risks** capture the probability that a flaw or a bug will impact the purpose of the software.
    - Risk = probability x impact
- A very high risk is not only likely to happen but also likely to cause great harm.
- **Vulnerabilities** are errors that an attacker can exploit.
    - Either flaws in the design or flaws in the implementation.

Bugs	Flaws
Buffer overflow: stack smashing	Method over-riding problems (subclass issues)
Buffer overflow: one-stage attacks	Compartmentalization problems in design
Buffer overflow: string format attacks	Privileged block protection failure (`DoPrivilege()`)
Race conditions: TOCTOU	Error-handling problems (fails open)
Unsafe environment variables	Type safety confusion error
Unsafe system calls (`fork()`, `exec()`, `system()`)	Insecure audit log design
	Broken or illogical access control (role-based access control [RBAC] over tiers)

Examples of Bugs and Flaws

Software security goals

- CIA (Confidentiality, Integrity, and Authentacation)

SDLC

1- Analysis
2- Design
3- Implementation
4- Testing
5- Maintenance

Lecture 2 : SDLC

Secure Software Design VS. Software Engineering

•Software Engineering (SE)

•Software Engineer

•Secure Software Design. Why?

•SE + Security engineering = secure software

Problems of producing secure software

•Attacker's knowledge

•Complexity of software

•Security education

•Attitude of software engineers

•Inadequacy of computer security models

Software development life cycle (SDLC)

The **SDLC phases** comprise the inception up to the retirement of the software product. These **phases** include **six** major **steps**: requirements and analysis, designing, implementation, testing, deployment, and maintenance.

systematic process, known as a system life cycle, which consists of six stages: feasibility study, system **analysis**, system design, programming and **testing**,

installation, and operation and **maintenance**. The first five stages are system development proper, and the last stage is the long-term exploitation.

1) Requirement gathering and analysis: Business requirements are gathered in this phase. This phase is the main focus of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements like; Who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system? These are general questions that get answered during a requirements gathering phase. After requirement gathering these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied.

Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model. The testing team follows the Software Testing Life Cycle and starts the Test Planning phase after the requirements analysis is completed.

2) Design: In this phase the system and software design is prepared from the requirement specifications which were studied in the first phase. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

In this phase the testers comes up with the Test strategy, where they mention what to test, how to test.

3) Implementation / Coding: On receiving system design documents, the work is divided in modules/units and actual coding is started. Since, in this phase the code is produced so it is the main focus for the developer. This is the longest phase of the software development life cycle.

4) Testing: After the code is developed it is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase. During this phase all types of functional testing like unit testing, integration testing, system testing, acceptance testing are done as well as non-functional testing are also done.

5) Deployment: After successful testing the product is delivered / deployed to the customer for their use.

As soon as the product is given to the customers they will first do the beta testing. If any changes are required or if any bugs are caught, then they will report it to the engineering team. Once those changes are made or the bugs are fixed then the final deployment will happen.

6) Maintenance: Once when the customers starts using the developed system then the actual problems comes up and needs to be solved from time to time. This process where the care is taken for the developed product is known as maintenance.

SSDM(Secure Software Design Method)

SSDM security engineering :

- Security training Security awareness: security concepts, breaches.
- Knowledge of attackers on previous related applications: previous attacks, tools, tech, virus…
- Understanding attacker's interest on the sw being developed.
- Knowledge about secure development practices. Previous practices strength weaknesse .
- Identify attackers and their capabilities.
- Divided into 3 parts

–Understand the nature of the software (type, complexity, other associating sw).

–Identify attackers/threats

–Identify possible vulnerabilities Security specification

•Security needs

•State security policies

•How to coordinate security implementation

•How to make the system to adapt to the changing landscape of security

•How to monitor security postures of the software system

Review security specification

•Design = security specification ?

Penetration testing

•Test the security of the sw and its resources

•Determine if the current security posture of the software is actually detecting and preventing attacks.

Software security vs. application security

•Software security.

•Application security.

Lecture 3 :The Three Pillars of Software security

Discussion

•Why software security must be part of a full life cycle approach? [HW]

Ans: **Security** is **important** to a **software development** company. Not only do they need to protect the **software** they **develop** and create, they also need to ensure the **security** of data that each and every one of their users generate and input

•Software security is building security functionality, integrating security features + software? [HW].:

Explaination :Software security—the process of designing, **building** and testing ... **Integrating** a decent set of best practices into the **software**

•Software security is not security software?[HW]

Ans: **Software security** is the idea of engineering **software** so that it continues to function correctly under malicious attack. ... **Software** defects with **security** ramifications—including implementation bugs such as buffer overflows and design flaws such as inconsistent error handling—promise to be with us for years

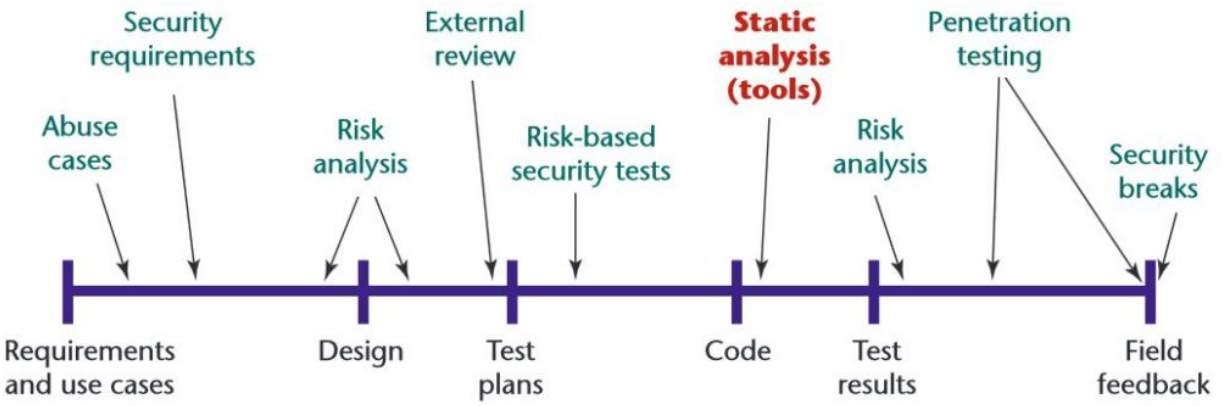


Figure 1. The software development life cycle. Throughout this series, we'll focus on specific parts of the cycle; here, we're examining static analysis.

The terms "application security" and "software security" are often used interchangeably. However, there is in fact a difference between the two. Information security pioneer Gary McGraw maintains that application security is a reactive approach, taking place once software has been deployed. Software security, on the other hand, involves a proactive approach, taking place within the pre-deployment phase.

To ensure that a piece of software is secure, security must be built into all phases of the software development life cycle (SDLC). Thus, software security isn't application security—it's much bigger.

Software doesn't recognize sensitivity or confidentiality of data that it is processing or transmitting over the Internet. Thus, software needs to be designed and developed based on the sensitivity of the data it is processing. If data is classified as "public," then it can be accessed without requiring the user to authenticate. One example is information found within a website's contact page or policy page. However, if the software performs user administration, then a multi-factor authentication method is expected to be in place to access this information. Based on classification of the data being processed by the application, suitable authentication, authorization, and

protection of data in storage or transit should be designed for the application in addition to carrying out secure coding.

To protect the software and related sensitive data, a measurement should be taken during each phase of the SDLC. This measurement broadly divides issues into pre and post-deployment phases of development. Again, software security deals with the pre-deployment issues, and application security takes care of post-deployment issues.

Software security (pre-deployment) activities include:

- Secure software design
- Development of secure coding guidelines for developers to follow
- Development of secure configuration procedures and standards for the deployment phase
- Secure coding that follows established guidelines
- Validation of user input and implementation of a suitable encoding strategy
- User authentication
- User session management
- Function level access control
- Use of strong cryptography to secure data at rest and in transit
- Validation of third-party components
- Arrest of any flaws in software design/architecture

Application security (post-deployment) activities include:

- Post deployment security tests
- Capture of flaws in software environment configuration
- Malicious code detection (implemented by the developer to create backdoor, time bomb)

- Patch/upgrade
- IP filtering
- Lock down executables
- Monitoring of programs at runtime to enforce the software use policy

•Bug
•Flaw
•Defect
•Software artifact

Pillars of software security

Pillar I: Risk management

•Tracking and mitigating risk as a full life cycle activity.

•A continuous risk management process is an essential part to software security.

•It identifies, ranks, tracks, and understands software security risks.

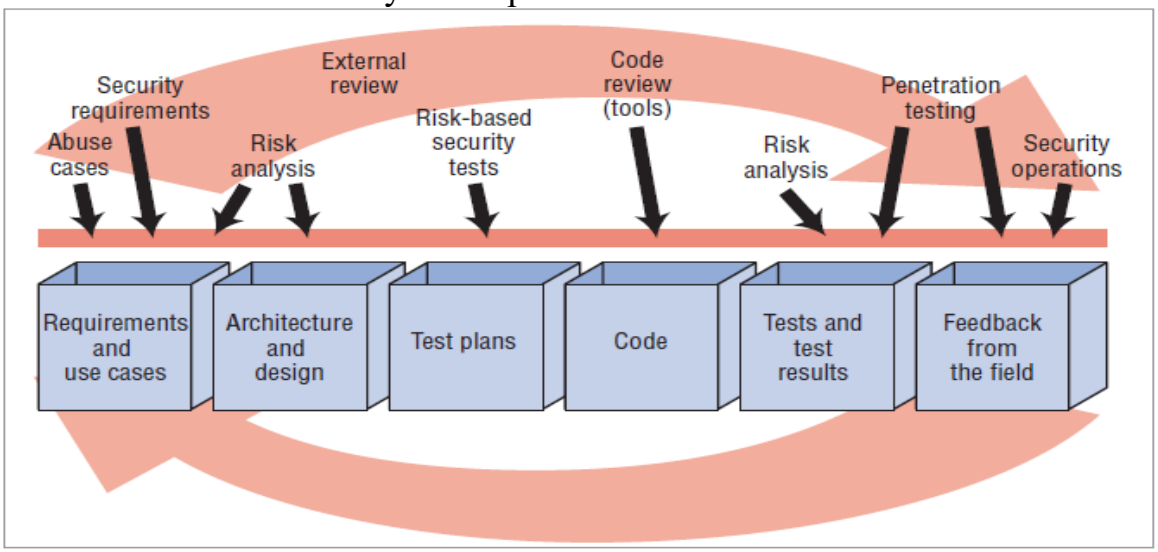
Pillar II: Software Security Touchpoints

•Software artifact. (requirements, design documents, code, test results).

•Applied to the software artifact produced through the SDLC.

•Appeared in IEEE magazine 2004 adopted by the US government, by Cigital, …

Pillar II: Software Security Touchpoints



•Applied regardless of the sw development process (e.g. waterfall model).

•SDLC + Touchpoints = SDL.

Pillar III: knowledge

•Involves gathering, encapsulating, and sharing security knowledge.

•Knowledge vs. information.{HW}

•Can be organized into 7 catalogs (principles, guidelines, rules, vulnerabilities, exploits, attack patterns, historical risks).

•Grouped into 3 categories (prescriptive knowledge, diagnostic knowledge, historical knowledge).

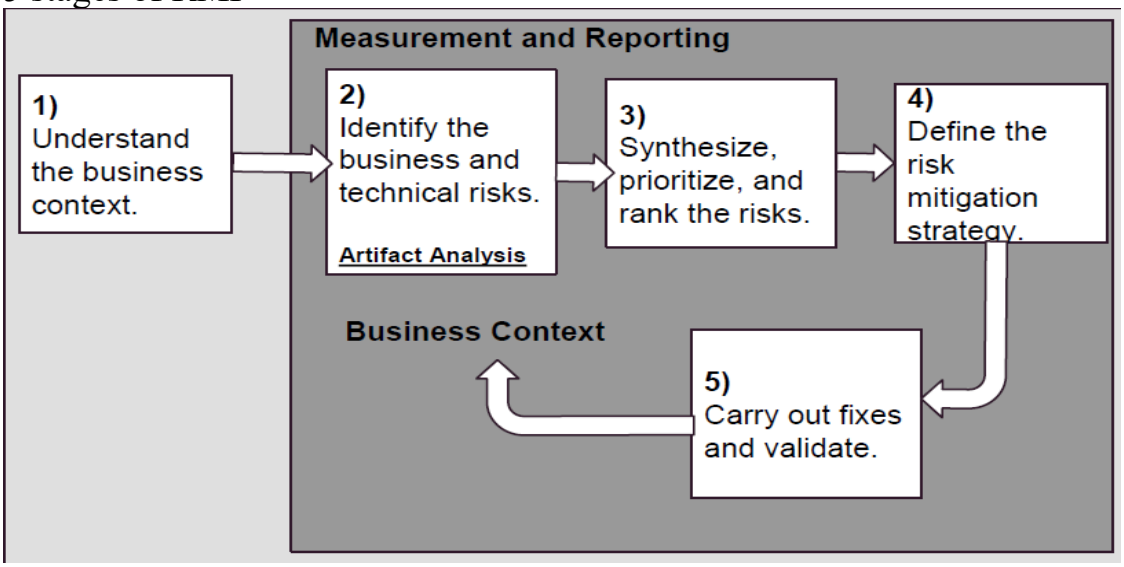
•These catalogs supporting software security.

•Knowledge can be applied at various stages throughout the entire SDLC.

•One way is through the use of software security touchpoints.

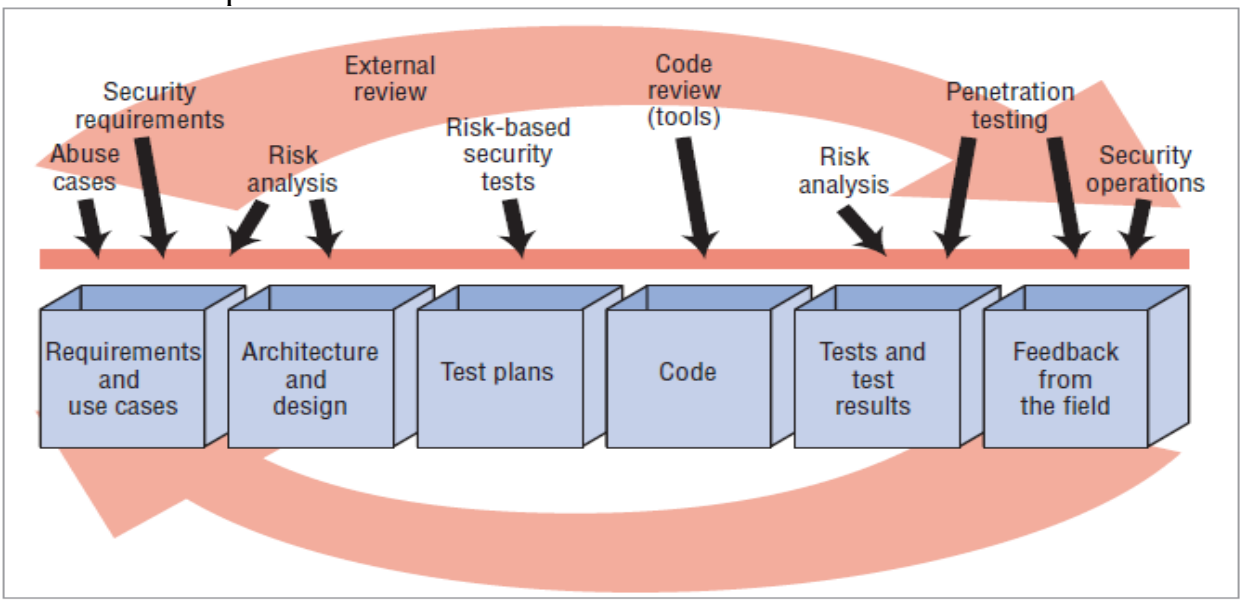
Pillar I:Risk Management
•Risk management framework (RMF)
•An overall approach to risk management.
•The goal is to consistently track and handle risks

5 stages of RMF



RMF activities
•Risk management is a central software security practice.
•Successful use of RMF relies on continuous and consistent identification of risks.
•Use project management tools to track risk information.
–Example: Open Workbench.
•RMF is a multilevel loop.
–Identifying risks only once during the project is incorrect.
–The five fundamental activities need to be applied repeatedly throughout the project.

Pillar II: Touchpoints



Seven touchpoints

1. Code Reviews.
•Artifact: Code.
•Example of risks found: Buffer overflow on line 30.

2. **Architectural Risk Analysis.**
•Artifact: Design and specifications.
•Example of risks found: Failure of a Web Service to authenticate calling code.

3. **Penetration Testing.**
•Artifact: System in its environment.
•Example of risks found: Poor handling of program state in Web interface

4. **Risk-Based Security Testing.**
•Artifact: Units and system.
•Example of risks found: Extent of data leakage possible by leveraging data protection risk.

5. **Abuse cases.**
•Artifact: Requirements and use cases.
•Example of risks found: Susceptibility to well-known tampering attack.

6. **Security Requirements.**
•Artifact: Requirements.

•Example of risks found: Explicit description of data protection needs is missing.

7. **Security Operations.**
•Artifact: Fielded system.
•Example of risks found: Insufficient logging to prosecute a known attacker

Fixing defects

1. Code review
•Source code analysis.
•Focus is on finding and fixing bugs.
•Use static analysis tools to find security bugs.
–Static analysis tools suffer from *false negatives and false positives.* [HW]
–Examples: Coverity, Fortify, Ounce Labs, and Secure Software.
•Static Analysis Tool ?
•Static analysis tools suffer from false negatives and false positives ?
•Solutions can be found in ch4 of reference [1].

2: Architectural Risk Analysis
•Detecting design flaws.
•50% of security problems design flaws.
•We can not detect it at the code level.
•Analysts uncover and rank architectural flaws.
•Link system concerns
 probability & impact measures. {risk=pr*impact}

Examples of Flaws lead to security risks
•Examples of design-level problems include
•error handling in object-oriented systems,

•object sharing and trust issues,
•unprotected data channels (both internal and external),
•incorrect or missing access control mechanisms,
•lack of auditing/logging or incorrect logging.

Risk-Based Security Testing
•Security testing must encompass two strategies:

(1) testing of security functionality with
standard functional testing techniques and
(2) risk-based security testing based on attack
patterns, risk analysis results, and abuse cases.
A good security test plan embraces both strategies.

Abuse Cases // Misuse Cases
•Building abuse cases is a great way to get into the mind of the attacker.
•Similar to use cases, abuse cases describe the system's behavior under attack; building abuse cases requires explicit coverage of what should be protected, from whom, and for how long.
•"What might some bad person cause to go wrong here?"

Security Requirements
•Security must be explicitly worked into the requirements level.
•Good security requirements cover both:
1. Functional security (say, the use of applied cryptography)
2.Emergent characteristics (best captured by abuse cases and attack patterns).

Penetration Testing

•Penetration testing is the most frequently and commonly applied of all software security best practices.

•It is a late-lifecycle activity and can be carried out in an outside in manner

Penetration testing vs. Security testing

•Penetration testing is by definition an activity that happens once software is complete and installed in its operational environment.

•security testing can be applied before the software is complete, at the unit level, in a testing environment with stubs and pre-integration

End

Lecture Four:

Deadly Sins (defects) of software security Buffer Overflow

Overview:

Security Defects:

• We live in an age with constant threat of security breaches

– Holes in web software

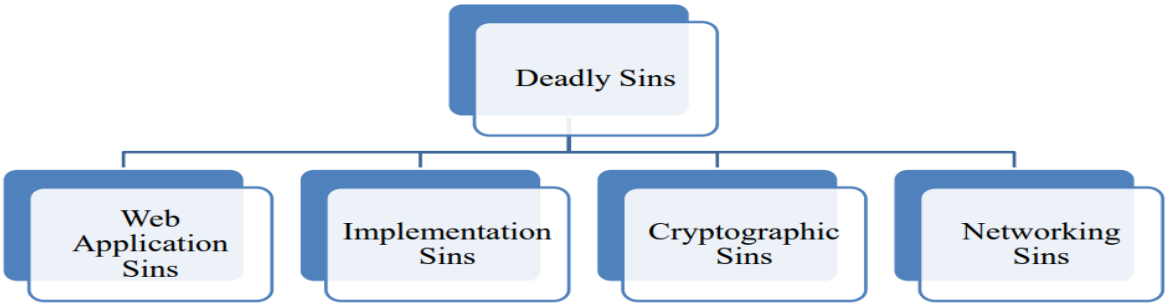
– Flaws in server software

• Security defects very easy to make

– Blaster worm defect only two lines long

– One line error can be catastrophic

• Here we look at 19 common security defects (sins of security)



19 Deadly sins

1. Buffer Overruns
2. Format String Problems
3. Integer Overflows
4. SQL Injection
5. Command Injection
6. Failing to Handle Errors
7. Cross-Site Scripting

8. Failing to Protect Network Traffic
9. Use of “magic” URLs and Hidden Forms
10. Improper use of SSL and TLS
11. Use of Weak Password-Based Systems.
12. Failing to Store and Protect Data Securely.
13. Information Leakage.
14. Improper File Access.
15. Trusting Network Name Resolution.
16. Race Conditions.
17. Unauthenticated Key Exchange.
18. Cryptographically Strong Random Numbers.
19. Poor Usability.

- **Sin 1: Buffer Overruns**

• You’ve heard this one many times>>>

• Occurs when a program allows input to write beyond the end of the allocated buffer

– Program might crash or allow attacker to gain control

– Still possible in languages like C#, Java since they use libraries written in C/C++ but more unlikely

• “Smashing the Stack” “ Buffer Overflow”

• The core problem is that user data and program flow control information are intermingled for the sake of performance, and low level languages allow direct access to application memory.

• Mostly prevalent in C / C++

• Occurs when a program allows input to write beyond the end of the allocated buffer.

• The effect of a buffer overrun is anything from a crash to the attacker gaining complete control of the application.

Buffer Overruns Examples:

Example (I)

```
#include Void DontDoThis(char* input)
{
char buf[16];
strcpy(buf, input);
printf("%s\n", buf);
}
Int main(intargc, char* argv[])
{
DontDoThis(argv[1]);
return 0;
}
```

```
0x0012FEC0  c8 fe 12 00  Èþ.. <- address of the buf argument
0x0012FEC4  c4 18 32 00  Ä.2. <- address of the input argument
0x0012FEC8  d0 fe 12 00  Ðþ.. <- start of buf
0x0012FECC  04 80 40 00  .[] @.
0x0012FED0  e7 02 3f 4f  ç.?O
0x0012FED4  66 00 00 00  f... <- end of buf
0x0012FED8  e4 fe 12 00  äþ.. <- contents of EBP register
0x0012FEDC  3f 10 40 00  ?.@. <- return address
0x0012FEE0  c4 18 32 00  Ä.2. <- address of argument to DontDoThis
0x0012FEE4  c0 ff 12 00  Àÿ..
0x0012FEE8  10 13 40 00  ..@. <- address main() will return to
```

Example (II)

```
// Robert T. Morris finger worm in 1988
char buf[20];
gets(buf);
char buf[20];
fgets(buf, num, stdin) // use fgets
```

Example (III)

```
#define MAX_BUF 256
void BadCode (char* input)
{
short len;
char buf[MAX_BUF];
len = strlen(input);
```

```
if (len< MAX_BUF) strcpy(buf, input);
}

constsize_t MAX_BUF = 256;
void LessBadCode (char* input)
{
size_tlen;
char buf[MAX_BUF];
len = strnlen (input, MAX_BUF);
 if (len< MAX_BUF) strcpy(buf, input);
}
 If short is 2bytes, input > 32767 ⟶ negative len // problem1
If input is not null-terminated // problem2
```

Example (IV)

```
 char buf[20];
 char prefix*+=“http://”;
strcpy(buf, prefix);
strncat(buf, path, sizeof(buf));
```

Spotting the sin pattern

Here are the components to look for:

- Input, whether read from the network, a file, or the command line.
- Transfer of data from said input to internal structures.
- Use of unsafe string handling calls.
- Use of arithmetic to calculate an allocation size or remaining buffer size

Redemption steps

- Replace dangerous string handling functions.
- Audit allocations.
- Check loops and array accesses.
- Replace C string buffers with C++ strings.
- Replace static arrays with Standard Template Library (STL) containers.
- Use analysis tools.

➢ Sin 2: Format String Problems

- The root cause of format string bugs is trusting user-supplied input without validation.

• Prevalent in C / C++ • Can also occur when the format strings are read from an untrusted location the attacker controls

Format String Examples :

Example(I)

#include int main(intargc, char* argv[])
{
if (argc>1) printf(argv[1]);
return 0;
}

"%x %x" 12ffc0 4011e5 %x: reads the stack 4 bytes at a time and outputs them Leaks important information to the attacker

Example (II)

#include int main(intargc, char* argv[])
{
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Your input was %d characters long\n", bytes);
return 0;
}

"Hello" Hello Your input was 5 characters long The %n specifier writes 4 bytes at a time based on the length of the previous argument Allows an attacker to place his data onto the stack

- **Spotting the Sin Pattern**

• In C / C++ look for functions from the printf family. Look for the following problems: – printf(user_input); – fprintf(STDOUT, user_input);

- **Redemption Steps**

• Never pass user input directly to a formatting function, and also be sure to do this at every level of handling formatted output.
• C / C++ redemption: printf("%s", user_input);

- **Sin(4):SQL Injection**

• How do bad guys get credit card numbers from sites?

– Break into server using exploit like buffer overrun
– Go through open port with sysadmin password
– Social engineering – SQL injection attacks

SQL Injection Example

- PHP code

```
$id = $_REQUEST[“id”];
 $pass = $_REQUEST[“password”];
$qry = “SELECT ccnum FROM cust WHERE id = $id AND pass=$pass”;
```

- PHP code

```
 $id = $_REQUEST[“id”];
$pass = $_REQUEST[“password”];
$qry = “SELECT ccnum FROM cust WHERE id = ‘$id’ AND pass=‘$pass’”;
```

User inputs id of user to attack For password, enters: ‘ OR 1=1 –
-- is the comment operator, to ignore whatever comes afterwards
Another:
Password:’ OR ‘’=‘

➢ Sin(18):Cryptographically Strong Random Numbers

• Seeds for pseudo-random number generators may not be that difficult to regenerate, then use to test a sequence of random values and determine what the next “random” number will be
• Can try true random number generators – Mouse, keyboard, Random.org, etc

Buffer Overruns Summary

• Do carefully check your buffer accesses by using safe string and buffer handling functions.
• Do use compiler-based defenses such as /GS and ProPolice.
• Do use operating-system-level buffer overrun defenses such as DEP and PaX.
• Do understand what data the attacker controls, and manage that data safely in your code.
• Do not think that compiler and OS defenses are sufficient—they are not; they are simply extra defenses.
• Do not create new code that uses unsafe functions.
• Consider updating your C/C++ compiler since the compiler authors add more defenses to the generated code.

• Consider removing unsafe functions from old code over time.
• Consider using C++ string and container classes rather than low-level C string functions.

SQL Injection Summary

• Do understand the database you use. Does it support stored procedures? What is the comment operator? Does it allow the attacker to call extended functionality?
• Do check the input for validity and trustworthiness.
• Do use parameterized queries, also known as prepared statements, placeholders, or parameter binding to build SQL statements.
• Do store the database connection information in a location outside of the application, such as an appropriately protected configuration file or the Windows registry.
• Consider removing access to all user-defined tables in the database, and granting access only through stored procedures. Then build the query using stored procedure and parameterized queries.

Cryptographically Strong Random Numbers Summary

• Do use the system cryptographic pseudo-random number generator (CRNGs) when at all possible.
• Do make sure that any other cryptographic generators are seeded with at least 64 bits of entropy, preferably 128 bits.
• Do not use a noncryptographic pseudo-random number generator (noncrytographic PRNG).

Lecture 5: BUFFER OVERFLOW

- Introduction

One of the more advanced attack techniques is the buffer overflow attack

Buffer Overflows occurs when software fails to sanity check input it's been given

Where more input is given than room set aside, you have an overflow. If the input is just garbage, most of the time the program will simply crash. This is because when the buffer gets filled, it can step on program code

- How Buffer Overflow Occur?

When a function is executing, it needs to store data about what it is doing. To do this, the computer will provide a region of memory called a stack

When a function is called, it needs to make sure that the stack has enough room for all of its data

If there is not enough room, the function will not be able to use all of its data, and an error will occur

If the function does not realize that there isn't enough room, the function may go ahead and store the data regardless-overwriting the available stack and corrupting it-usually crashing the program

Smashing : The Stack Smashing the stack is the terminology for being able to write past the end of a buffer and corrupt the stack

The stack is a contiguous block of memory in which data is stored

When the stack is smashed, or corrupted, it is possible to manipulate the return address and execute another program

Hackers have often write a simple code (called payload) to overflow the buffer

- What Happens When Buffer is Overflowed

Once you know that you have a working buffer overflow, you need to find out what part of your buffer is being used to load the instruction pointer

When you built your buffer, simply encode it with a predictable pattern

Once a stack overflows is successful, the return address from a function calls pushed onto the stack; a buffer overflow can overwrite the value

One of the central challenges to designing a good buffer overflow is finding a new address to overwrite the original

The address must enable the attacker to run his or her payload

- **Methods To Execute Payload**

Methods To Execute Payload

✓ **Direct Jump**

Direct jumps means that you have told our overflow code to jump directly to a location in memory

First, the address of the stack may contain a NULL character, so the entire payload will need to be placed before the injector. So, it will limit the available size for your payload

Second, the address of your payload is not always going to be the same. This leaves you guessing the address you wish to jump to

✓ **Blind Return**

The ESP (stack pointer) registers points to the current stack location

Any ret instruction causes the topmost value on the stack to be popped into EIP (instruction pointer), and EIP now points to a new code address; this is called popping

If the attacker can inject initial EIP value that points to ret instruction, the value stored at ESP will loaded into ESI (source index)

A whole series of techniques use the processor registers to get back to the stack

- ✓ **Pop Return**

If the value on the top of the stack does not point to within the attacker's buffer, the injected EIP can be set to point to a series of pop instructions, followed by a ret instruction

This will cause the stack to be popped a number of times before a value is used for the EIP register

The attackers just pop down the stack until the useful address reached

This method was used in at least one public exploit for Internet Information Server (IIS)

- ✓ **Call Register**

If a register is already loaded with an address that points to the payload, the attacker simply needs to load EIP to an instruction that performs a "call edx" or "call edi" or equivalent (depending on the desire register)

- ✓ **Push Return**

Uses the value stored in a register

If the register is loaded, but the attacker cannot find a "call" instruction, another option is to find a "push, " followed by a return

- **Designing Payload**

Designing Payload

Coding the payload Injection VectorLocation of Payload

- ✓ **Coding the payload**

There is a better way to encode payload. Just simply write them in C, C++ or inline assembly.

Then, copy the compiled code directly into the payload

It is easy for integrating assembly and C while using most compilers

It is called Fusion Technique

It is simple to encode and compile assembly language using Fusion Technique

- ✓ **Injection Vector**

The injection vector is the custom operational code to own the instruction pointer on the remote machine

The purpose of injection vector is to get the payload to execute

- ✓ **Location of Payload**

The location of the payload mustn't be same place with the injection vector

It is important to care about how the injection vector interacts with the payload

Now : Payload can be stored somewhere. It is to get the processor to start executing that buffer.

The common places to store payloads are:

1**. Files on the disks, which are then loaded into memory**

2. Environment variables controlled by a local user

3. Environment variables passed within a web request

4. User-controlled fields within a network protocol

- ## Finding New Buffer Overflow Exploits

First step in discovering a new buffer overflow is to insert invalid data into an application

To begin, allocate every point where data is accepted into a program

Secondly, the best overflows are often those that are injected through TCP/IP

- **How To Protect Buffer Overflow**

❖ **Choice of programming languages**

❖ **Use of safe-libraries**

❖ **Stack-smashing protection**

❖ **Executable space protection**

❖ **Address space layout randomization**

❖ **Deep packet inspection**

Lecture 6:Secure Software Design

Using the SafeStr Library

• **Problem**: You want an alternative to using thestandard C string-manipulation functions to helpavoid buffer overflows, format-string problems,and the use of unchecked external input.

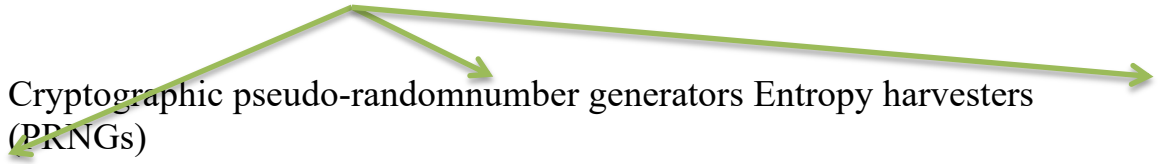
• **Solution**: use the SafeStr ("Safe Strings Library")

SafeStr function	C function
safestr_append( )	strcat( )
safestr_nappend( )	strncat( )
safestr_find( )	strstr( )
safestr_copy( )	strcpy( )
safestr_ncopy( )	strncpy( )
safestr_compare( )	strcmp( )
safestr_ncompare( )	strncmp( )
safestr_length( )	strlen( )
safestr_sprintf( )	sprintf( )
safestr_vsprintf( )	vsprintf( )

Random Number Generators

The need for random numbers:
–Key generation.

Determining what kind of randomnumbers to use
• There are three different kinds of random number generators



Cryptographic pseudo-randomnumber generators (PRNGs)

Entropy harvesters

Insecure random numbergenerators

First type:Insecure random numbergenerators:
• These are non-cryptographic pseudorandomnumber generators.
• should generally assume that anattacker could predict the output of such generator.

Second type: Cryptographic pseudo-randomnumber generators (PRNGs)
• These take a single secure seed andproduce as many unguessable random numbers from that seed as necessary.Such a solution should be secure for most uses as long as a few reasonableconditions are met (the most important being that they are securely seeded).

Third type:Entropy harvesters
• These are sometimes "true" randomnumber generators—although they really just try to gather entropy from other sources and present it directly. They are expected to be secure under mostcircumstances, but are generally incredibly slow to produce data.

Now, Implementing cryptographicpseudo- randomness function

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/rand.h>
unsigned char *spc_rand(unsigned char *buf,size_t l)
{
if (!RAND_bytes(buf, l))
{
fprintf(stderr, "The PRNG is not seeded!\n");
abort( );
}
returnbuf;
}
```

Getting random integers

```
unsignedintspc_rand_uint(void)
{
unsignedint res;
spc_rand((unsigned char *)&res,
sizeof(unsigned int));
return res;
}
```

Getting random integer in a range

```
#include <limits.h>
#include <stdlib.h>
intspc_rand_range(int min, int max)
{
unsignedintrado;
int range = max - min + 1;
if (max < min) abort( );
 /* Do your own error handlingif appropriate.*/
do
{
rado = spc_rand_uint( );
} while (rado> UINT_MAX - (UINT_MAX %range));
return min + (rado % range);
}
```

Or by using the following:

```
#include <stdlib.h>
intspc_rand_range(int min, int max)
{
if (max < min) abort( );
return min + (spc_rand_uint( ) % (max -
min + 1));
}
```

Or by using the following:

```
#include <limits.h>
intspc_rand_range(int min, int max)
{
if (max < min) abort( );
return min + (int)((double)spc_rand_uint( ) *
(max - min + 1) / (double)UINT_MAX)
% (max - min);
}
```

Note:H.W

- **Write a piece of program to do a random floating point value.**

Getting a Random Printable ASCIIString

```
#include <stdlib.h>
char *spc_rand_ascii(char *buf, size_tlen)
{
char *p = buf;
while (--len)
*p++ = (char)spc_rand_range(33, 126);
*p = 0;
returnbuf;
}
```

Input Validation

• **Sources of input:** files, command line(shell)
• **Input validation**: defensive technique.
• Assume some people send **malicious data**to our software.

Now the question :-
• What does our application do with that data?

To answer this question do the following :

-Understanding basic data validationtechniques
• Basic rules for proper data validation:
– **Assume all input is guilty until provenotherwise**.
• you should never trust external input that comes from outside the trusted base.
• you should be very skeptical about which components of the system are trusted, even after you have authenticated the user on theother end!
– **Prefer rejecting data to filtering data**.
– **Perform data validation both at input points and at the component level**.
• (**defense in depth** you should provide multiple defenses against aproblem if a single defense may fail)
• You can check the validity of data as it comes in from the network,and you can check it right before you use the data in a manner that might possibly have security implications.
– **Do not accept commands from the user unless you parse them yourself**.
• If the component doing the parsing has to trust its caller, bad thingscan happen if your software does not do the proper checking.
– **Beware of special commands, characters, and quoting**.
• One obvious thing to do when using a command language such as the Unix shell or SQL is to construct commands in trusted software,instead of allowing users to send commands.
– **Make policy decisions based on a "defaultdeny" rule. (white listing& Black listing).**
There are two different approaches to data filtering: white listing, you accept input as validonly if it meets specific criteria. Otherwise, you reject it. If you do this, the major thing youneed to worry about is whether the rules that define your white list are actually correct!

blacklisting, you reject only those things that are known to be bad. It is much easier to get your policy wrong when you take this approach.

– **You can look for a quoting mechanism, but know how to use it properly**
• you can usually stick untrusted data in single quotes
• you need to be aware of ways in which an attacker can leave the quoted environment.
• what happens if the attacker puts a single quote in thedata? Will that end the quoting, allowing the rest of theattacker's data to do malicious things?
– **When designing your own quoting mechanisms, do not allow escapes.**
• Provide functions that properly quote an arbitrary piece of data for you.
For example, you might have a function that quotes a string for a database, ensuring that the input will always be interpreted as a single string and nothing more.
– **The better you understand the data, the better you can filter it**
• Even if you filter out all bad characters, are the resulting combinations of benign characters a problem?
• The best way to ensure that data is not bad is to do your very best to understand the data and the context in which that data will be used. Parse the data as accurately as possible before sending.

Validating Email Addresses

• Email address: someone@gmail.com

• **Problem**: your program accepts an email address as input and you need to verify that the supplied address is valid.

• **Solution:** scan the email supplied by the user, and validate it against a lexical rules.

• If anyone attempts to use one of these no-longer relevant address formats, you can be reasonably certain they are attempting to do something they are not supposed to do.

Preventing SQL injection attacks

• **Problem:** You are developing an application that interacts with a SQL database, and you need to defend against SQL injection attacks.

• **Example**:

SELECT * FROM people WHERE first_name="frank";

SELECT * FROM people WHERE first_name="frank"; DROP TABLEpeople;

• Solution: The best way to avoid SQL injection attacks is to not create SQL command strings that include any user input. Avoid including user input in SQL commands as much as you can, but where it cannot be avoided, you should escape dangerous characters.

• There are two main approaches that can be taken to avoid SQL injection attacks:

1. Restrict user input to the smallest character set possible, and refuse any input that contains characteroutside of that set. Avoid “, ‘, %, _

2. Escape characters that have special significant to SQLcommand processors.
   - You should always enclose literals with quotes(“ or ‘).
   - Characters that should always be escaped are control charactersand the escape character itself (a backslash).

• if you are using the LIKE keyword in a WHERE clause, you may wish to prevent input from containing wildcard characters (% , _ , [,])