Introduction to Software Security

Mohammad Ghafari
Software Composition Group
Fall 2016
Overview

- Security mindset
- CIA triad
- Secure Development Lifecycle
- Buffer overflow attack and the countermeasures
Why Security

- **Asset:** lots of sensitive data (individuals, business and government)

- **Threat:** adversaries and criminals can economically or politically abuse this data
Security Mindset

- **Threats**: who are the bad actors?
- **Vulnerabilities**: what can possibility they exploit?
- **Risk**: if threats succeed to exploit a vulnerability, what is that attack/risk going to be?
Example: vulnerability

Is there any vulnerability in this bike?
Example: attack

Vulnerability that we never thought was that we had to secure more than just a wheel. In fact, the lock only protected a wheel not the whole bike.
Software Security
The Threat Landscape 2016

- Beyond 430 million unique pieces of malware existed in 2015 (36 percent more than 2014);
- A new zero-day vulnerability was discovered, on average, once each week in 2015. (125% increase from the year before);
- Above %75 of all legitimate websites have unpatched vulnerabilities that can be exploited. And, %15 of them are critical vulnerabilities.

Symantec report: https://resources.ekis.symantec.com/LP-2099
An example of Nation-states is the attack to the Sony pictures company in late 2014, when the Sony wanted to release “The Interview” movie that didn’t show the North-Korean leader in positive light.
Vulnerability and Attack

- **Vulnerability**: The weak points in software that can lead to security concerns.

- **Attack**: When threats uncover the vulnerability, conduct research about it, and exploit it to launch their schemes.

https://nvd.nist.gov/reports/different-common-vulnerabilities
The alternatives

- Make threats go away
- Reduce vulnerabilities
- Strive to meet security requirements of sensitive information:
  - Confidentiality
  - Integrity
  - Availability

CIA triad is not the Central Intelligence Agency
Confidentiality

- Encryption
- Access control (rules and policies)
  - Based on identity (e.g., name, serial number), role (e.g., manager)
- Authentication
  - password, card, fingerprint, iris scanner
- Authorization
- Physical security
  - Locked windowless rooms, Faraday cages

Confidentiality: data cannot be disclosed to unauthorized parties.
Integrity

- Backup
- Checksum
- Data correcting codes

Integrity: no one (except the authorized people) should be able to change the data.
Availability

- Physical protection
- Computational redundancies

Availability: the data is critical in the sense what we use it for is critical. If the data goes away we can’t do something that is really important to us.
Example

- Confidentiality:
  - encryption of traffic data,
  - time out for invalid inputs,
  - return invalid card, retain
  - stolen card, use of TAN in
  - net-banking

- Integrity:
  - consistency of data during
  - transmission

- Availability:
  - diverse network, fair
  - resource sharing
We strive to meet security requirements using various mechanisms but we can’t guarantee that there is no vulnerability in the system.

What should we do

- Prevention (see upcoming slides)
- Detection
- Response
- Recovery
The role of software security

- Software security is not only about reactive technologies like firewalls, intrusion detection systems, and antivirus engines.

- Remember that "software security" is not always "security software".

See the book "Software Security" by Gary McGraw

A security software helps to keep things secure (e.g., crypto libraries, authentication and authorization sub-systems, firewalls, and etc.) but software security is the property of software (an engineering discipline) that allows it to continue to operate as expected even when it is under attack.
There seems to be an ongoing misconception that good software security is not compatible with tight schedules. Following a roadmap, i.e., Secure Development Lifecycle one can ensure that secure software can take its place as a top priority along with software features and delivery timelines.
Delivering a secure software requires that all members of a software development team (developers, testers, and program managers) receive appropriate training, at least once a year, to stay informed about core security concepts.

Training

All stakeholders should stay informed about security basics and recent trends in the field.

Core secure trainings:
- Threat modeling
- Secure design, coding, and testing
- Privacy
- Advanced stuff, if resource available

Delivering a secure software requires that all members of a software development team (developers, testers, and program managers) receive appropriate training, at least once a year, to stay informed about core security concepts.
Considering security and privacy during the initial planning stage enables integration of the concerns in a way that minimizes disruption to plans and schedules.

Requirements

- Security requirements
  - Specifying minimum requirements
  - Specifying and deploying a vulnerability tracker
- Quality gates/bug bars
  - Defining criteria for acceptable level of security
- Risk assessment
  - Identifying the functional aspects of the software that requires deep review.
Actions to describe how to securely implement all the features that will be directly exposed to the users.

**Attack Surface** is set of ways in which adversaries can attack a software system. ASR is a means of reducing risk by giving attackers less opportunity to exploit a potential weak spot or vulnerability.

**Threat modeling** is the primary security analysis task performed during the software design stage to consider, document, and discuss the security implications of designs in the context of their planned operational environment.
Use approved tools by the project security advisor to take advantage of new security analysis functionality and protections.

Deprecate unsafe functions and replace them with safer alternatives.

Apply static analysis to ensure that secure coding policies are being followed.
**Verification**

- Dynamic analysis
- Fuzz testing
- Threat model and ASR review

**Dynamic analysis** is to verify if the program’s functionality works as designed.

**Fuzz testing**: is a specialized form of dynamic analysis used to induce program failure by deliberately introducing malformed or random data to an application.

**Threat model and ASR review** of a given application when its code is complete. This review ensures that any design or implementation changes to the system have been accounted for, and new risks have been reviewed and mitigated.
Even programs with no known vulnerabilities at the time of release can be subject to new threats that emerge over time. **Final security review** usually includes an examination of threat models, exception requests, tool output, and performance against the previously determined quality gates or bug bars.

- **Passed FSR**: all security and privacy issues identified by the FSR process are fixed or mitigated.
- **Passed FSR with exceptions**: Security advisor and team can reach an acceptable compromise about any SDL requirements that the project team was unable to resolve. These remaining requirements will be corrected in the next release.
- **FSR with escalation**: if a team does not meet all SDL requirements and the security advisor and the product team cannot reach an acceptable compromise, they escalate to higher management for a decision.

**Release and archive**: The security advisor assigned to the release must certify that the project team has satisfied security requirements. All pertinent information and data must be archived to allow for post-release servicing of the software.
Protect customers from software security or privacy vulnerabilities that emerge by executing the incident response plan.
The top five vulnerabilities in terms of potential for catastrophic damage based on the recent report by Infosec institute.
Buffer Overflow is the most common security vulnerability in OS. Here we focus on stack based buffer overflow and omit many details.
When buffer overflows the program would crash but we can skillfully exploit it to e.g., make data corrupt, steal some private information or run our arbitrary code.

---

**Basic concepts**

- **Buffer** is a contiguous block of computer memory that holds multiple instances of the same data type (like array in C).

- **Buffer Overflow** happens when one puts more data in a buffer that it can handle.
Address Space

- **Text**: machine code of the program, compiled from the source code.
- **Data**: static program variables initialized in the source code before execution.
- **BSS**: static uninitialized variables.
- **Heap**: data dynamically generated during the execution.
- **Stack**: structure that keeps track of the activated methods, their arguments, and local variables.
When a program starts running in the memory, the stack always points to the very highest memory address that it can use. Assume the instruction pointer (IP) is pointing to the `sum` method call in the `main` method.

```java
int sum(int a, int b){
    int sum;
    sum = a + b;
    return sum;
}

void main(){
    sum(10,5);
    // next instruction
}
```
A function call is found, push parameters on the stack (in reverse order from right to left). So, 5 will be pushed first and then 10.

```c
int sum(int a, int b){
    int sum;
    sum = a + b;
    return sum;
}

void main(){
    sum(10,5);
    // next instruction
}
```
We need to know where to return after function `sum` call is completed, so push the address of the next instruction on the stack.

```c
int sum(int a, int b){
    int sum;
    sum = a + b;
    return sum;
}

void main(){
    sum(10,5);
    // next instruction
}
```
Find the address of “sum” function and set IP to that value. The control has been transferred to the “sum” function.

As we are in a new function we need to update “BP” (to know in which function the execution happens). Before updating we save it on the stack so we can return later back to the main (the function we were before calling a new function). So, BP is pushed on the stack.

Now set BP to be equal to SP. Now, BP points to current function.
Push local variable `sum` onto the stack.
Each register is 4-bytes (when the binary is compiled for a 32-bit operating system)

When sum gets over, we need to back to previous stack frame.
[1] Set SP back to BP.
[2] Pop the earlier BP from stack and store it back in BP register
(execution returns to statement after the call)
Changing the return address -> We change the next to be executed instruction that can belong to the original code or our own arbitrary code
In computer security, a NOP sled is a sequence of no-operation instructions meant to "slide" the CPU's instruction execution flow to its final, desired, destination.

If the program jumps to anywhere into the sled, it will run all the remaining NOPs, doing nothing, and then will run the shell-code, just next to the sled.
Buffer Overflow Exploit

```c
void main(int argc, char** argv){
    char buffer[500];
    strcpy(buffer, argv[1]);
    return 0;
}
```

```
buffer | BP | return | ...
```

```
exploit | shell code | return
```

```c
/x90.../x90
```
Is there any BufferOverflow vulnerability in this C code?
The answer is yes. Normally, strlen() does not count the null terminator at the end of the string so if the buffer is of size 500 then null termination will be written beyond the maximum length, 500. Therefore even if buff == 500 the attack could happen.
Take a **missile** as an analogy. You have the rocket and fuel and everything else in the rocket, and then you have the warhead that does the actual damage. The delivery system (missile) is the exploit and the payload (warhead) is the (shell) code that actually does something.
In some cases it's not easy to find room for your shell code on the stack and get it executed (e.g. the system does not allow the execution). There is a variation of Stack Overflow attack called return-to-libc in which a subroutine return address on a call stack is replaced by an address of a subroutine that is already present in the memory.

Therefore, if you return to the right kind of library function and setup the arguments for it, then you can execute it with arguments of your choice and for example launch a command shell.
When control transfers to shell code, what privilege are going to be used when attacker code get executed?

Program runs on behalf of some users. So, attacker gets the same privilege as the host program during execution (e.g., can be a system process with root access).
Example
There are a number of defenses that help us counter attacks that rely on buffer overflow.

Countermeasures

- Safe program level
- Unsafe program level
- OS level
Protection (Safe Program level)

- Using strongly typed languages
- Automatic memory management and bound checking
  - Range checking: check that a number is within a certain range.
  - Index checking: check that in all expressions indexing an array the index value is checked against the bounds of the array.

C# and Java are examples of safe languages in which buffer overflow does not take place.
Protection (Unsafe Program level)

• Input checking
• Use safer functions that do “bounds checking”
  – e.g., think of “bounded string copy” in which a
    programmer should specify the number of
    characters to copy.
• Use automatic tools that analyze your code
  (source or executable) and warn about potential
  unsafe code fragments
  – e.g., see a list of them at https://www.owasp.org
**Stack canaries**: when a return address is stored in a stack frame a particular value, called canary, associated to that return address is written somewhere in the memory (usually just before it). Any attempt to rewrite the return address using buffer overflow will likely result a different value for the canary and overflow will be detected.

**ASLR (Address Space Layout Randomization)**: It randomizes the memory address space so that attackers cannot easily find the key locations in the memory e.g., libc function address.
Summary

- The meaning of security mindset;
- The notion of confidentiality, integrity, and availability for data protection;
- Different phases of Secure Development Lifecycle;
- The concepts behind the buffer overflow attack;
- And, the common protection techniques.