Recap

In the last lecture I covered some of the types of computing system security issues, a bit about the “hackers” and what/why they do the things they do.

In this lecture I want to concentrate on software security.

Specifically, what are some of the major attack vectors, and how can we make software more secure against these attacks.

I also want to talk a little bit about using encryption to provide protection, and look at a case study regarding encryption.

Security Software vs Software Security

Computer security software or cybersecurity software is any computer program designed to enhance information security.

This IS NOT what I plan to lecture about!

What does Software Security Mean?

• Software security is an idea or technique which is implemented to protect software against malicious attack and other hacker risks so that the software continues to function correctly under such potential risks.

• Security is necessary to provide
  – confidence
  – integrity
  – availability

Definition

• Techopedia explains Software Security:
  – Any compromise to confidentiality, integrity, and availability makes a software unsecure.
  – Software systems can be attacked to steal information, monitor content, introduce vulnerabilities and damage the behavior of software.
  – Malware can cause DoS (denial of service) or crash the system itself.

Types of Software Problems

• Buffer overflow, stack overflow, command injection and SQL injections are the most common attacks on the software.

• Buffer and stack overflow attacks overwrite the contents of the heap or stack respectively by writing extra bytes.

• Command injection can be achieved on the software code when system commands are used predominantly.
  – New system commands are appended to existing commands by the malicious attack.
  – Sometimes system command may stop services and cause DoS.
Types of Software Problems

- SQL injections use malicious SQL code to retrieve or modify important information from database servers.
  - SQL injections can be used to bypass login credentials.
  - Sometimes SQL injections fetch important information from a database or delete all important data from a database.
- The only way to avoid such attacks is to practice good programming techniques.
  - System-level security can be provided using better firewalls.
  - Using intrusion detection and prevention can also aid in stopping attackers from easy access to the system.

Buffer Overflow

- A buffer overflow occurs when more data are written to a buffer than it can hold.
- The excess data is written to the adjacent memory, overwriting the contents of that location and causing unpredictable results in a program.
- Buffer overflows happen when there is improper validation (no bounds prior to the data being written).
- It is considered a bug or weakness in the software.

Buffer Overflow

- Attackers can exploit a buffer overflow bug by injecting code that is specifically tailored to cause buffer overflow with the initial part of a data set, then writing the rest of the data to the memory address adjacent to the overflowing buffer.
- The overflow data might contain executable code that allows the attackers to run bigger and more sophisticated programs or grant themselves access to the system.
- Volume 7, Issue 49 of the Frack Journal (1996) contained a very good paper called “Smashing the Stack for Fun and Profit”.
  - Written by “Aleph One” (aka Elias Levy) this paper covered how to find/exploit buffer overflows.
  - http://phrack.org/issues/49/14.html#article

Buffer Overflow

- Many things have changed since The release of the “Smashing the Stack” paper.
  Linux (and other operating systems) have added system level protections in an attempt to make it more difficult to use the buffer overflow method.
  Some chip vendors have added hardware changes which also attempt to make buffer overflow exploits more difficult.
  But, alas, buffer overflows are still possible!

Buffer Overflow

- Buffer overflows are one of the worst bugs that can be exploited by an attacker mostly because it is very hard to find and fix, especially if the software consists of millions of lines of code.
  - Even the fixes for these bugs are quite complicated and error-prone.
  - That is why it is really almost impossible to remove this type of bug entirely.
- Although all programmers know the potential threat of buffer overflow in their programs, there are still a lot of buffer overflow-related threats in both new and old software, regardless of the number of fixes that have already been performed.

Buffer Overflow example

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

int main(void)
{
    char buff[15];
    int pass = 0;
    printf("Enter the password : 
");
    gets(buff);
    if(strcmp(buff, "thegeekstuff"))
    {
        printf("Wrong Password 
");
    } else {
        printf("Correct Password 
");
        pass = 1;
    }
    if(pass) {
        /* Now give root or admin rights to user*/
        printf("Root privileges given to the user 
");
    }
    return 0;
}
```

At the left is a contrived example of a buffer overflow.

The user is asked to enter a password in order to obtain administrator privileges.

If the given password is the same as the expected password, the program will give the user administrator privileges.

But if a wrong password is provided, and is longer than 15 characters, what is going to happen?
Buffer Overflow example

If a proper password of length < 15 characters is provided the program works as expected:

$ ./bfrovrflw
Enter the password :
thegeekstuff
Correct Password
Root privileges given to the user

If an incorrect password of length > 15 characters is provided, we get an unusual result:

$ ./bfrovrflw
Enter the password :
hhhhhhhhhhhhhhhhhhhh
Wrong Password
Root privileges given to the user

Buffer Overflow

In the previous example we managed to overflow the buffer (the memory allocated to the buf variable), and write over the value of the integer variable “pass”.

If we enter a 30 character password, what will happen?
In all likelihood, we will get a segfault.

• Why? Because we have 15 bytes (in reality probably 16 bytes) allocated to “buf”, and 4 bytes allocated to the integer pass (for a total of 19/20 bytes)...but we entered 30 bytes.
• We just wrote over portions of memory which we do not “own”, so the overflow is detected, and the system issues a segfault.

Buffer Overflow Mitigation

The best mitigation for buffer overflows is safe programming practices.
In addition, there are several applications available which will scan your source code and flag unsafe constructs/code segments so that you can examine the source and attempt to fix the issues.

– Coverity
– Veracode
– Stylecop
– Eclipse
– ...

Problems with C, C++, C#, php, ...

Many of the stock library routines that are part of the C family of languages are dangerous to use. For example:

scanf, fscanf, sscanf, gets, strcat, sprintf, strcpy, vsprintf, splitpath, makepath, getwd, ...

These routines DO NOT provide bounds checking.
Use bounds checked versions if one is available:

• strcpy -> strncpy -> strlcpy/strcpy_s
• strcat -> strncat -> strlcat/strcat_s -strtok
• sprintf -> snprintf
• vsprintf -> vsprintf
• gets -> fgets/gets_s

If you must use unsafe functions, make sure to do your own bounds checks!

Stack Overflow

In the buffer overflow example we managed to overflow a buffer without using a procedure call.

A stack overflow is a runtime error that happens when a program runs out of memory in the call stack (i.e.: we are calling a subroutine).

• The stack overflow generally signals a problem in resource provisioning and has to be fixed in order to allow the program to run and use memory properly.
• The “attack” idea is very similar to the buffer overflow…
  • But now we can overwrite the administrative data on the stack, and (hopefully) cause the system to return to our “shell code” instead of returning to the place the author of the program thought we would return to.
  • Once again, if we are successful, we get an administrator/root shell, and we own the system.

• Moral of the story: ALWAYS bounds check string operations!
main(int argc, char *argv[]) {
    func(argv[1]);
}

void func(char *v) {
    char buffer[10];
    strcpy(buffer, v);
}

Stack Overflow

- It is important to point out that a stack overflow is a logical runtime error and not a syntax error.
- Syntax errors result when the computer cannot read a section of code correctly, and these errors are caught by the compiler or at other key points before production.
- A stack overflow, on the other hand, happens at runtime and can be hard to figure out.

Stack Overflow Mitigation

Perhaps the most important single protection is one variously known as W^X ("write exclusive-or execute"), DEP ("data execution prevention"), NX ("No Xecute"), XD ("eXecute Disable"), EVP ("Enhanced Virus Protection," a rather peculiar term sometimes used by AMD), XN ("eXecute Never"), and probably more.

The principle here is simple. These systems strive to make memory either writeable (suitable for buffers) or executable (suitable for libraries and program code) but not both.

Thus, even if an attacker can overflow a buffer and control the return address, the processor will ultimately refuse to execute the shellcode.

Stack Overflow Mitigation

Operating system and compiler developers have implemented a number of systems to make exploiting these overflows harder.

Some of these systems are intended to make specific attacker tasks harder.

One set of Linux patches made sure that system libraries were all loaded at low addresses to ensure that they contained at least one null byte in their address; this makes it harder to use their addresses in any overflow that uses C string handling.

Other defenses are more general. Many compilers today have some kind of stack protection. A runtime-determined value known as a "canary" is written onto the end of the stack near where the return address is stored.

At the end of every function, that value is checked for modification before the return instruction is issued. If the canary value has changed (because it has been overwritten in a buffer overflow) then the program will immediately crash rather than continue.
Command Injection

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application.

Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell.

Command Injection 1

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    char cat[] = "cat ";
    char *command;
    size_t commandLength;
    commandLength = strlen(cat) + strlen(argv[1]) + 1;
    command = (char *) malloc(commandLength);
    strncpy(command, cat, commandLength);
    strncat(command, argv[1], (commandLength - strlen(cat)));
    system(command);
    return (0);
}
```

The code on the left allows a command injection attack to occur.

When used normally, the output is:

$ ./catWrapper Story.txt

When last we left our heroes...

When used maliciously:

$ ./catWrapper "Story.txt; ls"

When last we left our heroes...

Story.txt               doubFree.c              nullpointer.c
unstosig.c            www*                       a.out*
format.c               strlen.c                    useFree*
```
catWrapper*        misnull.c                  strlength.c
useFree.c            cmdinjection.c         nodefault.c
trunc.c                  writeWhatWhere.c
```

Command Injection 2

```c
int main(char* argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Any guesses what this piece of code will do if given the input:

```
";rm -rf /
```

This program is compiled, and installed setuid root.

Command Injection 3

```c
char* home=getenv("APPHOME");
char cmd[CMD_MAX] = "/usr/bin/cat";
strcat(cmd, argv[1]);
execl(cmd, NULL);
```

The code to the left was found in a piece of code that came with a commercial application.

It runs setuid root.

The code obtains the “home” directory of the application from the environment variable APPHOME, and then runs an init command to set up the user’s environment so they can run the app.

What happens if the user sets the APPHOME environment variable to a directory which contains their own INITCMD?

Command Injection

How can we protect against command injection?

One simple thing that we can do is to not accept any input which contains semicolons:

- Replace or Ban arguments with ";"
- But other shell escapes are available
  - Examples:
    - &&
    - |

These examples (and more) come from a paper at:
https://owasp.org/www-community/attacks/Command_Injection

And another one at:
https://portswigger.net/web-security/os-command-injection

Command Injection

By far the most effective way to prevent OS command injection vulnerabilities is to never call out to OS commands from application-layer code.

- In virtually every case, there are alternate ways of implementing the required functionality using safer platform APIs.

If it is considered unavoidable to call out to OS commands with user-supplied input, then strong input validation must be performed.

Some examples of effective validation include:

- Validating against a whitelist of permitted values.
- Validating that the input is a number.
- Validating that the input contains only alphanumeric characters, no other syntax or whitespace.
SQL Injection

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL Injection

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;

Look at the example above again. The original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 105 OR 1=1

Then, the SQL statement will look like this:

SELECT * FROM Users WHERE UserId = 105 OR 1=1;

SQL Injection Based on 1=1 is Always True

What if passwords are also stored in the user table?

Now the attacker has all user names and passwords!

SQL Injection Based on "="=" is Always True

What if a web site asks users to provide a login and password in order to access the site. The code for this might be:

uName = getRequestString("username");
uPass = getRequestString("userpassword");
sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'

And the SQL code passed to the database would be:

SELECT * FROM Users WHERE Name = "John Doe" AND Pass = "myPass"

But what happens if the user gives

Username: " or "="
Password: " or "="

The resulting SQL is:

SELECT * FROM Users WHERE Name ="" or ";" or "="

The SQL above is valid and will return all rows from the "Users" table, since OR ";" is always TRUE.

SQL Injection

SELECT * FROM Users; DROP TABLE Suppliers

Look at the following example:

txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;

And the following input:

User id: 105 ; DROP TABLE Suppliers

The valid SQL statement created would look like this:

SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;

SQL Injection

To protect a web site from SQL injection, you can use SQL parameters. SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.
SQL Injection

The examples on the previous pages came from:
https://www.w3schools.com/sql/sql_injection.asp

Another good reference for SQL Injection is:
https://owasp.org/www-community/attacks/SQL_Injection

Encryption

- As I have mentioned in previous lectures, encryption is one of the tools at our disposal that we can use to protect plain-text data from prying eyes.
- If we encrypt the data, interlopers cannot read the content, nor can they alter the content.
- A plain text from a user can be encrypted to a ciphertext, then sent through a communication channel and no eavesdropper can interfere with the plain text.
- When it reaches the receiver end, the ciphertext is decrypted to the original plain text.
- Unfortunately, this protection comes at a price.
  - Encryption requires more CPU/memory
  - We need to “share keys” with the recipients
  - We need to install utilities to handle the encrypted data.

- It should be pointed out that encryption will NOT protect against poor programming practices!
- There are two basic types of encryption utilities available to us:
  - Symmetric Encryption -
    - This is the simplest kind of encryption that involves only one secret key to cipher and decipher information.
    - Symmetrical encryption is an old and best-known technique.
    - It uses a secret key that can either be a number, a word or a string of random letters. It is a blended with the plain text of a message to change the content in a particular way.

- The sender and the recipient should know the secret key that is used to encrypt and decrypt all the messages.
- Blowfish, AES, RC4, DES, RCS, and RC6 are examples of symmetric encryption.
  - NOTE: Some of these algorithms are known to be insecure!
- The most widely used symmetric algorithm is AES-128, AES-192, and AES-256.
- The main disadvantage of the symmetric key encryption is that all parties involved have to exchange the key used to encrypt the data before they can decrypt it.
Encryption

- Asymmetric Encryption
  - Asymmetrical encryption is also known as public key cryptography, which is a relatively new method, compared to symmetric encryption.
  - Asymmetric encryption uses two keys to encrypt a plain text.
    - Secret keys are exchanged over the Internet or a large network.
      - It ensures that malicious persons do not misuse the keys.
      - It is important to note that anyone with a secret key can decrypt the message and this is why asymmetrical encryption uses two related keys to boosting security.
    - A public key is made freely available to anyone who might want to send you a message.
    - The second private key is kept a secret so that you can only know.
  - A message that is encrypted using a public key can only be decrypted using a private key, while also, a message encrypted using a private key can be decrypted using a public key.
  - Security of the public key is not required because it is publicly available and can be passed over the Internet.
  - Asymmetric key has a far better power in ensuring the security of information transmitted during communication.
  - Asymmetric encryption is slower than symmetric encryption.
  - Asymmetric encryption is mostly used in day-to-day communication channels, especially over the Internet.
  - Popular asymmetric key encryption algorithm includes ElGamal, RSA, DSA, Elliptic curve techniques, PKCS.

Digital Certificates

- To use asymmetric encryption, there must be a way of discovering public keys.
  - One typical technique is using digital certificates in a client-server model of communication.
  - A certificate is a package of information that identifies a user and a server.
    - It contains information such as an organization's name, the organization that issued the certificate, the users' email address and country, and users public key.
  - When a server and a client require a secure encrypted communication, they send a query over the network to the other party, which sends back a copy of the certificate.
    - The other party’s public key can be extracted from the certificate.
    - A certificate can also be used to uniquely identify the holder.
  - SSL/TLS uses both asymmetric and symmetric encryption, quickly look at digitally signed SSL certificates issued by trusted certificate authorities (CAs).

The symmetric encryption algorithms are generally considered "stronger" than the asymmetric encryption algorithms.

But sharing the keys for symmetric encryption is a much more difficult issue than sharing keys for asymmetric encryption.

We use asymmetric encryption every day – sometimes without knowing that we did so.

Some applications which use asymmetric encryption:
  - ssh
  - https
  - Many email servers use it to provide authentication
  - gpg/pgp (Gnu Privacy Guard / Pretty Good Privacy)

So how else might we use encryption?

- Encrypt email so only the recipient can open it.
- Encrypt text messages
- Encrypt data which we store in a database
- Encrypt the disk in our laptop/desktop
- Encrypt important files

And if we are bad guys

- Encrypt the files on someone else’s system and demand a ransom in order to allow them to decrypt the files
  - Ransomware
- Encrypt our virus/worm code so it cannot be easily detected
Encryption gotcha

It’s estimated that within the next 10-15 years, there will be a quantum computer powerful enough to break existing RSA and ECC cryptographic algorithms, rendering most existing cryptography insecure. Most recently, IBM successfully demonstrated a quantum-proof encryption method it developed.

Encryption Gotcha

The entire reason for using encryption is to protect data from the eyes of others that we feel have no reason to view the data. Recently the use of encryption became a big topic in the news media. Because suspected criminals had used encryption on their cell phone(s), the government requested that cell phone manufacturers provide a way for the government to decrypt information on the phones.

Responsible Encryption

Security enhancements to the virtual world should not make us more vulnerable in the physical world. We must find a way to balance the need to secure data with public safety and the need for law enforcement to access the information they need to safeguard the public, investigate crimes, and prevent future criminal activity. Not doing so hinders our law enforcement agencies’ ability to stop criminals and abusers in their tracks.

Responsible Encryption

Companies should not deliberately design their systems to preclude any form of access to content, even for preventing or investigating the most serious crimes. This puts our citizens and societies at risk by severely hindering a company’s ability to detect and respond to illegal content and activity, such as child sexual exploitation and abuse, terrorism, and foreign adversaries’ attempts to undermine democratic values and institutions, preventing the prosecution of offenders and safeguarding of victims.

Responsible Encryption

- Warrant-proof encryption defeats the constitutional balance by elevating privacy above public safety.
- Encrypted communications that cannot be intercepted and locked devices that cannot be opened are law-free zones that permit criminals and terrorists to operate without detection by police and without accountability by judges and juries.
- Responsible encryption is achievable.
  - Responsible encryption can involve effective, secure encryption that allows access only with judicial authorization.
  - Such encryption already exists.
  - Examples include the central management of security keys and operating system updates; the scanning of content, like your e-mails, for advertising purposes; the simulcast of messages to multiple destinations at once; and key recovery when a user forgets the password to decrypt a laptop.

Responsible Encryption

On December 2, 2015, 14 people were killed and 22 were injured in a terrorist attack in San Bernardino, CA. One suspect, Syed Rizwan Farook, owned an iPhone 5c which was used to organize and coordinate the attacks. Because the iPhone 5c has disk encryption, it cannot be decrypted without knowing the correct cryptographic key (which was generated by combining the user’s passcode with a key baked into the hardware). In the past, Apple has unlocked around 70 phones for the authorities. However, these older phones did not have disk encryption. For older phones with no encryption, Apple had already had a software version to bypass the unlock screen (e.g. Genius Bar fixes). Due to the difficulty in unlocking the iPhone, the FBI is citing the All Writs Act of 1789, which allows a court to require third parties’ assistance to execute a prior order of the court when “necessary or appropriate.”

Responsible Encryption

We have great respect for the professionals at the FBI, and we believe their intentions are good. Up to this point, we have done everything that is both within our power and within the law to help them. But now the U.S. government has asked us for something we simply do not have, and something we consider too dangerous to create. They have asked us to build a backdoor to the iPhone. Specifically, the FBI wants us to make a new version of the iPhone operating system, circumventing several important security features, and install it on an iPhone recovered during the investigation. In the wrong hands, this software — which does not exist today — would have the potential to unlock any iPhone in someone’s physical possession.
The implications of the government’s demands are chilling. If the government can use the All Writs Act to make it easier to unlock your iPhone, it would have the power to reach into anyone’s device to capture their data. The government could extend this breach of privacy and demand that Apple build surveillance software to intercept your messages, access your health records or financial data, track your location, or even access your phone’s microphone or camera without your knowledge.

We are challenging the FBI’s demands with the deepest respect for American democracy and a love of our country. We believe it would be in the best interest of everyone to step back and consider the implications.

1. iOS can be set to erase its keys after 10 incorrect passcode guesses. The FBI wants software with this feature disabled.
2. iOS imposes rate limiting after consecutive incorrect passcode guesses to slow down guessing. The FBI wants software that accepts an arbitrary number of guesses with no delays.
3. iOS requires individual passcodes be typed in by hand. The FBI wants a means to electronically enter passcodes, allowing it to automatically try every possible code quickly.

Are the FBI's demands reasonable? Is Apple's response justified? Do we have a right to encryption? Should technology companies provide the government with user data? Should technology companies provide tools for the government to bypass or circumvent security measures? When does the state's compelling interest override the individual's autonomy?

In this lecture I have provided examples of a few types of attacks against software. In addition, I have shown some of the techniques to mitigate these attacks. As software engineers/programmers we need to do our best to insure that our code is not susceptible to such attacks.