Program Security I

Tom Kelliher, CS 325

Feb. 22, 2010

1 Administrivia

Announcements

Assignment

Read 3.4–3.7.

From Last Time

Project 1 discussion.

Outline

1. Secure programs.
2. Non-Malicious program errors.
3. Viruses and other malicious program errors.

Coming Up

Program security II.

2 Secure Programs

1. At this level, we're addressing how a program handles security itself, not assistance it receives from the OS or other, external, security mechanisms.
2. What do we mean when we say a program is *secure*?
   (a) Requires much effort and time to cause a failure.
   (b) Program has been in use for a long time with no failures.
   (c) No faults and, therefore, no failures.
   (d) Meets the level of security specified in the requirements.
   (e) Other measures?

3 Non-Malicious Program Errors

Programmer had no malevolent intent. Perhaps just "clueless."

3.1 Buffer Overflows

1. General idea:
   (a) Overflow a buffer with a code segment.
   (b) Segment overwrites existing code, a return address, etc.
   (c) Original program runs your code as whatever user.
   (d) You now have the privileges of that user.

       If root, it's time to party.

2. How an we mediate this type of an error?

3.2 Lack of Mediation

1. Unvalidated, unchecked data values.

2. Garbage input values can lead to a program entering an invalid state, crashing, etc., resulting in denial of service.

   Several years ago, numerous standard Unix tools were fed garbage data. Nearly all crashed.

3.3 Time-of-Check to Time-of-Use; Incomplete Mediation

1. Data values are checked and validated, but can be changed before use.

   Synchronization problem.

2. General idea:

   (a) Submit data that will validate.

   (b) Validation begins and completes.

   (c) Change data.

   (d) Use data.

3. Causes:

   (a) Client/Server interactions where client-side does validation.

   What prevents a hacker from accessing the server interface directly?

   (b) Internal validation in which there is a delay between validation and use with the data left accessible.

   Example: bank account transactions queued up on a linked list prior to being applied to the database.

4. Result: Gain arbitrary unauthorized access.

5. Prevention: Copy data to a secure location; validate and use from there.

   Example: OS kernel copies system call parameters to its own memory space before validating and using.

4 Viruses and Other Malicious Program Errors

Programmer did have malevolent intent.

4.1 Background

1. Where does this stuff come from?

   (a) Potentially, any executable or source file you download.

   (b) Viruses, spyware.

   (c) Hacked open source repositories.

2. Types:

   (a) Virus: Attaches itself to executable and begins propagating.

   (b) Trojan horse: Contains unexpected functionality.

   "Innocent" example: Grand Theft Auto's Hot Coffee scene.

   (c) Logic bomb: Activates when certain condition achieved.

   (d) Time bomb: Activates at certain time.

   (e) Trapdoor: Allows unauthorized access.

(f) Worm: Propagates through network.

(g) Rabbit: Massive replication, leading to resource exhaustion and DOS.

4.2 A Few Examples

1. Root kits. Hide malevolent activity with "patched" versions of standard system utilities, such as `ls`, `ps`, `netstat`, etc.

2. The Morris worm.

   (a) First Internet worm, unleashed Nov. 2, 1988.

   (b) Not created to "cause damage," but a program flaw caused this to become a "rabbit," resulting in severe DOS problems.

   (c) Written by Robert Tappan Morris at Cornell; released at MIT for purposes of disguise. Morris is now an assoc. profât MIT.

   His father worked for the NSA.

   (d) Exploited flaws in 4 BSD:

      i. Tried to match encrypted passwords in `/etc/passwd` through a brute force attack on "common" passwords, then words in a local dictionary.

      Fixes: Use of shadow password file; password salts; enforced use of better passwords.

      ii. Buffer overflow in `fingerd` — `gets()`.

      iii. Debug trapdoor in `sendmail` — allowed command execution. At that time, `sendmail` commonly ran as root!

      iv. Trusted hosts in `rsh`.

   (e) Overview of infection mechanism, assuming a shell is running on target machine, or source machine has established an SMTP connection to target and is transmitting commands:

i. Establishment of bootstrap on target machine:

   A. Break-in via `rsh` and `.rhosts`.

   B. Commands sent to `sendmail` in DEBUG mode.

ii. Open socket on infecting machine for bootstrap to connect to.

iii. Bootstrap code connects to source machine (server) and authenticates.

   It retrieves binary copies of the server code for Sun and VAX architectures as well as source code.

iv. Across network connection, server would attempt to infect target. If successful, it would disconnect. If unsuccessful, it would remove evidence and disconnect.

v. The newly-created worm on the newly-infected host hides itself.

vi. Worm now attempts to determine host connectivity, through use of `netstat` and reading system host files.

vii. Worm chooses a set of hosts to attempt to infect.

viii. Worm attempts to infect other hosts, using:

   A. `rsh` via `.rhosts`.

   B. Overflow `fingerd`'s input buffer, causing `execve("/bin/sh", 0, 0)` to execute on VAXen.

   C. Use the `sendmail` trapdoor.

ix. Attempt to find easy hosts to connect to, searching for `.rhosts`, etc. files.

x. Tried password attacks. This, in conjunction with `.rhosts` and `.forward` files would lead to likely new hosts to infect.

For details, see `http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf`.