Secure programming

Stefan D. Bruda

CS 464/564, Fall 2023

SECURITY

- Why bother at all?
  - The Internet is not a secure place. Many people try to crack systems, and the network infrastructure is inherently insecure
- How secure is secure?
  - No computer can be ever totally secure
  - Security needs vary from case to case (e.g., your home computer vs. your bank's)
  - The more secure your system is, the more intrusive security becomes. You need to decide when your system will still be usable, and yet secure for your purposes
- But wait, what does this have to do with this course?
  - Just an introduction to the matter at hand
  - The point is, your servers should lay as good a basis as possible for secure computers
  - If your server is crackable, then so is the machine it runs on; even if the server is installed improperly, the risks should be minimized

Security issues

- Risks: An intruder may subvert the server, e.g., make it read or write files, delete critical data, or execute arbitrary code
- Threats: Several types of intruders:
  - The Curious wants to see what you have in there
  - The Malicious wants to bring down your system
  - The High-Profile Intruder cracks your system for boasting rights
  - The Borrowers wants to use resources you pay for (e.g., bandwidth)
  - The Leapfrogger wants to use your system to attack others
  - The Competition
- Vulnerabilities: what are the security holes in the system
  - This is where you, the programmer of servers, come in: to minimize these
  - If the system is full of vulnerabilities it will eventually go down (and bring your server down too), but do not let your server be the cause

Unprivileged servers

Vulnerabilities are greatly minimized if your server runs unprivileged

- A program inherits not only the open descriptors, but also the user it belongs to
- However, a server is usually launched by the init system, which is run for obvious reasons as `root` (user ID 0)
- Root privileges are also needed at startup
- Once the startup is complete very few servers need root privileges
- Therefore as soon as you can you should drop root privileges, i.e., change the user ID your program runs under to something else than 0:

  ```
  setuid(non-privileged-uid);
  ```
- Group privileges are also important, and thus they should be dropped too:

  ```
  setgid(non-privileged-gid);
  ```
- This is arguably the biggest security improvement of them all
- Typically, servers launch as root but then switch to special user IDs, created just for them and which have the minimum amount of privileges

Confined servers

- Servers should change the current working directory to a safe directory
- Even so, nothing prevents them to write to any other directory: all they have to do is to provide full paths to the files they want to access
- Sometimes you cannot do anything about it (whenever the server must access files all over the place)
- But sometimes your server needs files that are all located in a specific subtree of your file system
- If this is the case, then you should confine your server to that subtree

```
chroot(dir);
```

  - The effect: dir becomes the root directory of your server
  - For instance, after your program does `chroot("/var/lib/shfd")`, it will view the file `"/var/lib/shfd/shfd.log"` as `"/shfd.log"`
  - A file which is someplace else (say, `"/etc/passwd"`) is simply inaccessible
- Once you go into a "chroot jail" you can not get back (not even as root)
- Arguably the second biggest security improvement, but difficult to implement
  - All the shared libraries necessary for running the program must be available in the chroot jail

Why run as root in the first place?

- Some system calls have no effect if run unprivileged; they include `chroot`, `setgid`, and `setuid`
- A non-root program cannot bind to ports below 1024
- So your server must run as root at the very beginning, just to issue these calls and/or open its ports
- As a consequence, once your server drops root privileges, it cannot get them back
- In other words, the proper sequence of calls is:

```
chdir("/var/lib/shfd");
chroot("/var/lib/shfd");
// open master socket on port below 1024
setgid(99);
setuid(99);
```

- Of course, there are cases when you have to run your server as root all the way (and perhaps also outside any chroot jail)
  - Then the potential of harm is huge
  - You should be extra careful when programming such a server

VALIDATE ALL INPUT

- Some inputs are from untrustable users, so those inputs must be validated (filtered) before being used
  - You should determine what is legal and reject anything that does not match that definition
  - Example of illegal strings: `".."`, anything starting with `/`, control characters (too small ASCII values) and/or characters with the high bit set (too large ASCII values)
  - But validate, do not do the reverse (do not identify what is illegal and write code to reject those cases)!

  Strings: identify the legal characters or legal patterns and reject anything not matching that form

    - A character sequence may have special meaning to the program's internal storage format (e.g., a slash in the name of a file); check for these

  Numbers: limit all numbers to the minimum (often zero) and maximum allowed values

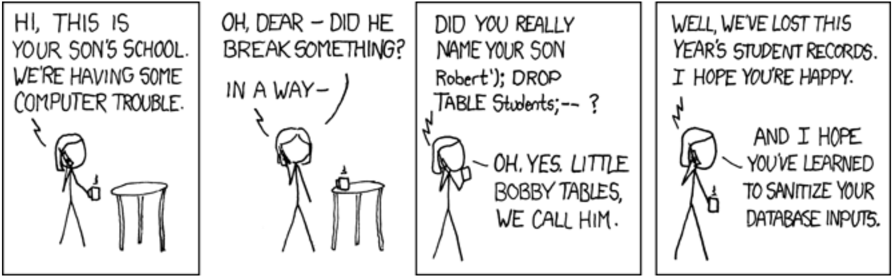
Validate all input (cont'd)

- Input includes but is not limited to command line arguments, environment variables, and things received from a client
  - Use text input as much as you can (easier to check)
- Limit the maximum character length (and minimum length if appropriate)
  - Be sure to not lose control when such lengths are exceeded
- Tests should usually be centralized in one place so that the validity tests can be easily examined for correctness later
- Make sure that your validity test is actually correct
  - This is particularly a problem when checking input that will be used by another program
  - These tests may have subtle errors, producing the deputy problem (the checking program makes different assumptions than the program that actually uses the data)

VALIDATE ALL INPUT (CONT'D)

- While parsing user input, it is a good idea to temporarily drop all privileges, or even create separate processes
  - This is especially true if the parsing task is complex, or if the programming language does not protect against buffer overflows (e.g., C and C++)
- Validate command line arguments
  - Attackers can send just about any kind of data through a command line (through calls such as `execve`)
  - You must definitely validate the command line inputs
    - In particular, never trust the name of the program reported by `argv[0]` (an attacker can set it to any value including NULL)
- Validate file descriptors
  - Do not assume that any file descriptor is opened and points to anything in particular
  - Better close them all and reopen what is needed (a matter of resource management but also of security!)

- Validate file names
  - Reject "globing" characters (`*`, `?`) whenever possible
    - If you must glob, do so in a separate process, with limits on resources
  - Filter dangerous file names, including:
    - Names beginning with a dash
    - Names with control characters (especially newlines) in them
    - Names containing spaces
    - Names containing characters with special meaning to the system and the programming language (e.g., <, `"`, `;`, etc.)
- Validate file content
  - If a program takes directions from a file, the file must be considered suspect unless only trusted users can control its content (meaning: untrusted users cannot modify the file, its directory, or any of its ancestor directories)
  - If the file is suspect, make sure that the inputs from the file are protected as described in other places (taking data from a file is not an excuse)



http://xkcd.com/327

AVOID BUFFER OVERFLOWS

- This is a very common and very dangerous security flaw
- When allocating data (e.g., an array), validate the size
  - It should be positive
- When accessing data in an array, validate the index
  - It should be within the array size, and positive
- When copying stuff, check for bounds and for the format of the output;
  - Especially important for strings
  - Use "safe" functions (e.g., `snprintf` instead of `sprintf`, `strncpy` instead of `strcpy`)
  - But do not forget that you may thus loose the terminating null byte!
- Avoid dangling pointers at all cost
  - Set deleted pointers to 0, and check before accessing the content of any pointer

The perrils of buffer overflow: A real-world example

- Sendmail debug flags: `-dflag,value`
  - "`sendmail -d8,100 ...`" sets flag number 8 to value 100
- Name of config file (`/etc/sendmail.cf`) also stored in memory (before the flags)
  - `/etc/sendmail.cf` gives the path to `/bin/mail`
- Sendmail checked for maximum flag numbers, but not for positiveness
- Integer larger than 2^{31} considered negative by C on 32-bit machines
- `sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113` changes "`etc`" to "`tmp`" in the name of the config file

		t	m	p																
...	/	~~e~~	~~t~~	~~c~~	/	s	e	n	d	m	a	i	l	.	c	f	\0	...	flag0	...

- Attacker then creates `/tmp/sendmail.cf` which claims local mailer is `/bin/sh`
  - debug call gives root shell!

RACE CONDITIONS

- Simple code gone wrong:

```
void incr() {
    x++;
}
```

Race conditions

- Simple code gone wrong:

```
void incr() {
    x++;
}
```

  - Three instructions (load `x`, increment register, store result), possibly executed in an interleaved manner → fails when called from multiple threads
  - Result depends on the interleaving = race condition
- Use synchronization primitives judiciously
- But also keep in mind that abusing critical regions can unboundedly decrease response time
- Choose carefulness instead of critical regions as much as possible, but do choose critical regions whenever applicable
- Race conditions can also happen because of signal handlers!

- Innocent code (typical producer-consumer system):

Blue thread	Red thread
`x = ...;`	`while (!done) {}`
`done = true;`	`... = x;`

Race conditions can be very subtle

- Innocent code (typical producer-consumer system):

Blue thread	Red thread
`x = ...;`	`while (!done) {}`
`done = true;`	`... = x;`

- ... Except that the compiler might obligingly break it for you!
  - Indeed, any optimization flag passed to the compiler might cause it to notice that `done` is not modified inside the loop
  - So the red thread might become `tmp = done; while (!tmp) {}`
  - ... or even `tmp = done; if (!tmp) while (true) {}`
  - Even if you don't pass any optimization flags the hardware might still optimize
  - This rather than CPU load is the reason why busy loops in user space are evil

RACE CONDITIONS (CONT'D)

- Many system calls are not thread safe (also called reentrant), that is, they can lead to race conditions when used concurrently
- Many such system calls have a "reentrant" variant (identified by the `_r` suffix), which is
  - Thread safe, but also
  - Harder to use and usually less efficient
- Use the reentrant variant whenever concurrent calls are possible, but use the normal variant when race conditions cannot happen
  - Tokenize a string which is a local (stack) variable → use `strtok`
  - Tokenize a string which is a global or heap variable → use `strtok_r`

Follow good principles for secure program

- Least privilege. Each user and program should operate using the fewest privileges possible, thus limiting the damage from an accident, error, or attack
- Economy of mechanism/Simplicity. The design of the protection system should be simple and small as possible; interfaces should be minimal, narrow, and non-bypassable; trust should be minimized
- Open design. The protection mechanism must not depend on attacker ignorance; the mechanism should be public, depending on the secrecy of relatively few (and easily changeable) items like passwords or private keys
- Complete mediation. Every access attempt must be checked; position the mechanism so it cannot be subverted; for instance, in a client-server model the server must do all access checking
- Fail-safe defaults. The default should be denial of service
- Separation of privilege. Ideally, access should depend on more than one condition, so that defeating one protection system won't enable complete access
- Least common mechanism. Minimize the amount and use of shared mechanisms (e.g. use of the `/tmp` or `/var/tmp` directories)
- Psychological acceptability/Easy to use. The human interface must be designed for ease of use so users will routinely and automatically use the protection mechanisms correctly

ONLY AN OVERVIEW

- This is just a brief incursion into security issues
- Other things that have strong impact on security:
  - Environment variables (they are very dangerous)
  - Random number generators
  - Etc.
- For more details about secure programming, take a look at

  `http://www.faqs.org/docs/Linux-HOWTO/Secure-Programs-HOWTO.html`

  and the references therein
- An instructive tutorial on buffer overflow exploitation:

  `http://www.cs.wright.edu/~tkprasad/courses/cs781/aleph0ne.html`