

Security Conscious Programming

Vincenzo Ciaschini

CNAF/CERN Security Seminars

02/12/2020

Contents

- A slim of what to do and not to do when writing a secure application.
 - Both from the design and code points of view.
 - All issues shown have been exploited in the wild
 - All the examples are real
 - All have already been fixed
 - All can still be exploited (in other ways)
 - Even if OS/hardware protections are in place

Design Issues

Control Dependencies

- Libraries are additional code that do not need to be recreated from zero.
- They reduce the need of writing new code. With new bugs.
- But they are code.
- And therefore can have bugs, even security bugs.
- Worse, they can be purposely compromised.
- Dependencies have dependencies themselves.
 - You may not even know which dependencies you have

Compromised Dependency example

- Example:
 - getcookies (JS Library)
 - A backdoor allowed injection of execution of code with a HTTP with and header including the string gCOMMANDhDATAi
 - COMMAND allowed to load or execute JS code
 - DATA was the code
 - Getcookies was used in several other programs. E.g.:
 - Mailparser -> http-fetch-cookies -> express-cookies ->getcookies

Suggestions on Dependencies

- Limit dependencies only to those absolutely necessary.
 - Do not use dependencies for trivial code like "justify text to the right"
- Always consider the FULL dependency tree
- Get dependencies for the distribution if at all possible
- If they MUST be downloaded from a different repository:
 - Never depend on just the master release.
 - Depend on a specific version. Keep abreast of updates, and promptly update the dependency when needed
 - Manually check the code and transitively the dependencies
- Limit dependencies only to those absolutely necessary.
 - Do not use dependencies for trivial code like "justify to the right"

Separation of interfaces

- Several webapps have both a user and administrator interface available.
- Sometimes they share a single code base and they are deployed at the same time on the same endpoints.
- Which means that access cannot be restricted
- Which means that an exploit on one interface can compromise the other

Interface compromise example

- Wordpress plugin vulnerability "Ultimate member" has a vulnerability that allows users to become a fully-fledged Wordpress administrator. (October 2020)
 - Could be exploited during User Registration to create a user that would have "administrator" role in Wordpress.

Suggestions on Separation of Interfaces

- If possible, separate the interfaces and make them deployable on different endpoints.
 - If the admin interface can be deployed separately, it can receive additional protections
 - For example, restricting the source of connections.
 - Even if the user-side interface is compromised, if write credentials are not there, they cannot be exploited.

Defense in Depth

- There is a tendency to delegate security to a single component that as the responsibility
- Which means that when that component fails or is sidestepped, there is no security at all.

Defence in Depth failure example

- CISCO wireless firewall with static credentials (July 2020) (CVE-2020-3330)
 - System account with default static firewall -> allowed full firewall takeover.
 - Everything that relied on that firewall for security was not secure at all.

Suggestions on Defense in Depth

- Implement security at all levels and in all components.
- If you have a gateway component, put security in the gateway, but also put checks in all the other components.
 - Assume that any protection will fail and put more protections in the line.
- The objective is to put as many obstacles in the way as possible

Coding Issues

User Input

- The best channel an attacker has is user input.
- Very few programs are useful without needing input from someone.
- But input can be weaponized to exploit many other kinds of vulnerabilities.
- So, all input should be checked for sanity before being used.

User Input check failure example

- Zzcms
 - Uses user input unchecked in SQL query (details in SQL injection section)
 - Allows attacker to do whatever it wants on the DB with the credentials of the application.

Suggestions for User Input

- Never trust it
- Always check for acceptability
 - e.g: If you are expecting a number, check that only numeric characters are present.
- Better yet, use it in a way that is safe even if the content is malicious. (examples later)

Memory corruption

- Applications that do not need buffers to keep information for a while are rare.
- Buffers have a length
- Writing past the length of a buffer may lead to things ranging from RCE to control flow subversion depending on specific and on OS/hardware protection active
- Also, if specific lengths are expected, always check that the exact length of the provided data is correct
- If using manual memory handling, never free memory twice, or use it after it has been freed.

Memory Corruption example

- Classic examples:
 - `char dst[10];`
 - `scanf("%s", dst);`
 - Where input longer than 9 char, or:
 - `char src[12];`
 - `strcpy(src, dst);`
 - Where size is exceeded
- Still happening today: Last year, OpenSSH 7 and 8, for example, had a Remote Code Execution caused by a similar problem. (unlikely to affect you, since it requires activating an experimental algorithm) or Firefox 65

Suggestions on Memory Corruption

- Always check the sizes of buffers and of the data you intend to copy in them BEFORE copying and fail if they do not match.
- If a specific size is expected, always check that that amount of data is actually present.
- If using c-like functions, always use the bounded variants (strncpy, strncpy, etc...) over the unbounded ones (strcpy...)
- If using manual memory handling, zero a pointer immediately after it has been freed.

Injection

- Input can be used to inject data inside an application:
 - Variants:
 - Injecting SQL in a query (ex: SQL Injection)
 - Injecting HTML or javascript in a webpage (ex: javascript injection/XSS)
 - Injecting code in an application (ex: serialization/deserialization)
- They all stem from the same cause: user-provided data utilized UNFILTERED in the program

Injection examples

- Zzmcs (CVE-2019-1010148):
- `if (isset($_REQUEST["img"])){`
- `$img=$_REQUEST["img"];`
- `}else{`
- `$img="";`
- `}`
- `if (isset($_REQUEST["oldimg"])){`
- `$oldimg=$_REQUEST["oldimg"];`
- `}else{`
- `$oldimg="";`
- `}`
- ...
- `if ($oldimg<>$img){`
- `$f=".." . $oldimg;`
- `if (file_exists($f)){`
- `unlink($f);`
- Any file can be deleted! Further code (not shown) allows this to become a code execution.

Suggestion to avoid Injection

- Never reuse data as-is.
 - In case of SQL, never construct query on-the-fly, but always use prepared queries.
 - When reflecting user input in the output, always use escaping functions before!
 - Do not roll your own! Use the OS- or framework-provided ones.
 - Do not load blobs in your code that you do not fully control!

Hardcoded or Default Credentials

- There is still much code around which hardcoded credentials, either in the code itself, or in its default configuration files
 - They may not necessarily be credentials to the application.
 - You can also find AWS bucket secrets, default keys
- They will not remain secret.

Examples of Harcoded or Default Credentials

- Many, many, many
- Just as examples, several CISCO networking apparatuses had hardcoded credentials that are periodically patched out with new software releases.
 - Routers, firewalls, etc...
 - Have fun losing control of your networking backbone

Suggestions on Harcoded or Default Credentials

- It is simple. Never use them.
- On config files, leave the fields empty, and have the application fail if they remain empty.
- On code, do not put them there to start with.
- If for some reason a set of initial credential is necessary, force changing them at first login, and make that first login a necessary step for configuration.

Credential Storage

- Software that uses local logins has a need to keep credentials.
- How they are stored matters!
- They should not be encoded in a form that can be reversed.
 - And especially not in plaintext.
- They can be leaked

Examples of Credential Storage

- Publish over Dropbox Jenkins Plugin (May 2019)
 - Stores secrets in plaintext in an xml file

Suggestions on Credential Storage

- Leak from applications are common.
 - Never store credentials in plaintext
 - Also do not store them encrypted
 - Encryption can be reversed by just a single additional leak
- Credentials should be stored in a non-recoverable form
 - For example, hashed and salted.
 - This will prevent reversion, and the salt will stop construction of rainbow tables

Logging Mistakes

- Logging by itself is far from a vulnerability.
 - But it may log too much or too little
- Logging too much examples:
 - Logging secrets from a failed (or succesfull!) login attempt
- Logging too little examples:
 - Not logging all errors, requests or failed login attempts
- Or unfiltered logging:
 - If logging info contains user input... See above slides about not trusting user input

Examples of Logging Mistakes

- A rather unusual examples: ANSI Escape Sequences
- `127.0.0.1 - - [10/Oct/2000:13:55:36 -0700] "GET / HTTP/1.0" 200 2326 "" "^[[1A"`
- Effect: when viewed on standard (ANSI-compliant) terminal, the sequence `ESC[1A` moves the cursor up one line. The next line of logs will be written over this one.
- Working very well to hide information

Suggestions on Logging Mistakes

- Always strip non-printable characters
- Do not log credentials
- Always log all requests
- Never log user input unfiltered
- Never make logs available from the net

Error Message Mistakes

- All applications handle error, and usually produce error messages.
- Some are terse, some are long, and some can be used to exfiltrate informations

Examples of Error Message Mistakes

- Example of an error message:
- ERR: testuser1topsecret123 logon failed(-1) in PASS
- This was a real error from a mail server when trying login as user testuser1
- Needless to say, topsecret123 (the password) was not known when starting the attack

Suggestions with Error Messages

- Be as vague as possible
 - As possible means give enough information to understand and correct the problem and not more. Examples:
 - When authenticating with a wrong password:
 - Wrong username/password - GOOD
 - Wrong password – WRONG
- Do not reflect user input unfiltered

....

- There are many other possible issues. This was just an introduction.
- Personal study is necessary to remain uptodate
- Some (more) generic suggestions in the following slides

Generic Suggestion

- Review your code (or let someone else review it) with an eye to ways to attack it
- Study some secure coding practices. Examples:
 - [CERT C++ Coding Standard](#)
 - There are versions for many languages
- Many interpreters or compilers have tools that make security checks automatically. Use them!