Secure Programming Techniques

Margus Freudenthal, Mart Oruaas

Lecture 7, 2022-04-01
Data Integrity
As Usual, Some Terminology 1/2

Useful model from “Why Programs Fail” [Zeller, 2005].

- **Defect**: an incorrect program code
- **Infection**: an incorrect program state, caused by a defect
- **Failure**: an observable incorrect program behavior, caused by propagation of an infection

See also [Dullien, 2017] for more formal discussion of the topic.
As Usual, Some Terminology 2/2

[Diagram of program states and input values showing transition from a sane state to an infected state, ending with an observer indicating a failure.]
Some Observations

- With security issues, input is carefully selected by the attacker to trigger the defects and cause infection.
- Input is **any bytestream** and **any other kind of signal** that reaches the running program.
  - This includes: user input, configuration files, data files, DNS responses, errors from downstream components, signals, interrupts (the list is not exhaustive).
- Often, the incorrect program state cannot be represented (and reasoned with) in your programming language.
  - As an example, consider buffer overflow in a Python program.
Some Examples
Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
    Cookie c = cookies[i];
    if (c.getName().equals("role")) {
        userRole = c.getValue();
    }
}
Relying on Untrusted DNS

```java
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);

if (addr.getCanonicalHostName().endsWith("trustme.com")) {
    trusted = true;
}
```
Downloading Widget from External Domain

Welcome!
Please Login:
Username: <input type="text" name="username"/>
<br/>
Password: <input type="password" name="password"/>
<input type="submit" value="Login" />

<script type="text/javascript"
    src="externalDomain.example.com/weather.js"/>

Lecture 7, 2022-04-01
Uploading Files

- There are different ways of determining file type, each one untrustworthy
  - File extension
  - MIME type
  - File contents (typically first few bytes)
- When saving a file, ensure that it can not be executed
- Ensure that you do not overwrite anything with the file
  - Meaning that you should probably ignore path information supplied by the user
- With compressed data, beware of the zip bombs
Integer Overflow

Example from OpenSSH 3.3

```c
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```
Deserialization

class VulnerableProtocol(protocol.Protocol):
    def dataReceived(self, data):
        # Code to actually parse incoming data
        # according to an internal state machine
        # If we just finished receiving headers, call
        # verifyAuth() to check authentication

    def verifyAuth(self, self, headers):
        try:
            token = cPickle.loads(base64.b64decode(
                headers['AuthToken']))
            if not check_hmac(token['signature'],
                token['data'], getSecretKey()):
                raise AuthenticationFailed
            self.secure_data = token['data']
        except:
            raise AuthenticationFailed
Cross-Site Request Forgery (CSRF)

- Main idea: user’s browser is directed to make request to some other host where the user is currently logged in
  - Because the request comes from browser, it already has all the session cookies
  - The main challenge for attacker is to ensure that the user has an open session with the victim site
- Simplest example: use GET URL:
  - Send in e-mail
  - Use as image href
  - Use HTTP redirect
  - Load from script
CSRF: POST Example

HTTP POST is a bit more tricky, but doable:

```html
<html>
  <body>
    <form action="https://victim.com/email/change" method="POST">
      <input type="hidden" name="email" value="pwned@evil-user.net" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```
CSRF: Mitigation

- Be suspicious about GET requests that change system’s state
- Use CSRF token
  - A random string that is tied to user’s session
  - Included (as hidden parameter) in the forms coming from the server
  - Not known to the attacker who has not received data from the victim server
  - For added security, can be different for every request
  - Most better frameworks have built-in support for CSRF tokens. Configure and use this feature
- For additional (incomplete) protection, use sameSite cookie attribute [OWASP, 2022]
private void buildList(int untrustedListSize) {
    if (untrustedListSize < 0) {
        Throw new IllegalArgumentException("Negative\list\size");
    }
    Widget[] list = new Widget[untrustedListSize];
    list[0] = new Widget();
}
Input Sanity Checking 2/2

- You should always sanity check your inputs with respect to your business logic
  - Negative prices and quantities
  - Absurdly big numbers
  - Typical technical issue: allocating too much memory/disk space based on declared data size (or sometimes also actual data size)

- Dive deep into your business logic and your state machine model
if (! propor->mode & PNG_HAVE_PLTE)) {
    /* Should be an error, but we can cope with it */
    png_warning(png_ptr, "Missing PLTE before tRNS");
} else if (length > (png_uint_32) propor->num_palette) {
    png_warning(png_ptr, "Incorrect tRNS chunk length");
    png_crc_finish(png_ptr, length);
    return;
}
...
png_crc_read(png_ptr, readbuf, (png_size_t)length);
Buffer Overflow: The Impact 1/2

Possible consequences of buffer overflows
- Reading (confidential) data from other variables
- Overwriting variables, causing logic errors
- Overwriting stack contents, remote code execution
Buffer Overflow: The Impact 2/2

Stack growth

Top of stack

Return address

Local variables

Buffer

Address increases

Attack code

Return address

Local variables

Buffer

Lecture 7, 2022-04-01
Buffer Overflow: Mitigation

Use a language that has memory safety

However, even with a safe language there may still be issues.
- Safe languages often have unsafe operations
- Much of the interesting stuff is still done using written in native code
- There may be issues with calling native code: type conversions, string representation, etc.
- The runtime itself may have vulnerabilities that manifest in some situations
Basics of Input Validation
Input Validation Goals

See [OWASP, 2021]

- “Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.”

- “Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.”

- “Input Validation should not be used as the primary method of preventing XSS, SQL Injection and other attacks [...] but can significantly contribute to reducing their impact if implemented properly.”
Syntactic Validation

- The aim is to enforce correct syntax of structured fields (SSN, date, ...) 
- Syntactic validation should be performed using a **parser** that is based on a solid formal language describing the input bytestream’s structure
- Regular expression is a form of formal grammar for regular languages, use it, but smartly!
- In reality RE-s cover most of the custom syntactic input validation needs of a business application (the rest can be left to good standard libraries)
- There is a whole movement dedicated to formally correct input validation: [http://langsec.org/](http://langsec.org/)
Semantic Validation

- The aim is to enforce correctness of input values in the specific business context (e.g. start date is before end date, price is within expected range)

- Failures in semantic validation can and will lead to direct business consequences (see: all recent cryptocurrency hacks)

- Example tools for tackling semantic validation:
  - Separate business rules (domain specific logic) from general execution flow, validate all parsed input via business rules
  - Maintain a good understanding of application’s business-level state model
  - Use everything available to enforce the business rules: rule engines, database transactions and constraints, etc.
Conclusion

- Make (security) decisions only based on trusted data
- How do you know it’s trusted?
  - It was generated by you and never left the system
  - It was generated by you and you verified that it came from you
  - It was generated by a trusted party (e.g., identity server, certification authority) and you verified it
  - You calculated this fact based on input data and a valid algorithm
Questions?
References I


References II

Cross-site request forgery prevention cheat sheet.
https://cheatsheetseries.owasp.org/cheatsheets/
Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

Morgan Kaufmann.