Session 2: How software breaks (2/2)

Part 1: Output encoding and tainting as defensive principles

A conceptual input / processing / output diagram, which is used in the following discussion.

- There are various strategies in which these injection attacks can be mitigated:
  - Remove all unnecessary functionality. All intended (correct) functionality comes with bugs and flaws, leading to unintended functionality. By removing all unnecessary functionality, you decrease the probability that you have introduced unintended functionality.
  - Restrict the input language to the simplest possible language (grammar). As an example, if your input can be easily described with a regular expression, validity checks are much easier to do. You might even want to consider a “front end” that modifies the input from a complex one into a simple one, and use sandboxing to restrict that “front end” component. This is the design used in, for example, the qmail MTA.
  - Drop all invalid input. Instead of trying to interpret the input, require it to conform to whatever specification you have, and if it is invalid, ensure that rejection creates the least potential performance impact and does not leak information.
about the processing back to the attacker (e.g., in side channels).
  ○ Treat all untrusted input as TAINTED until it has been UNTAINTED. As a practical example, use prepared statements and parameterised queries for SQL, where there is a static SQL template, which is populated by attacker-controlled query parameters (but because the template is static, they cannot affect the statement);
  ○ Ensure that everything is always encoded appropriately with regard to in which context it is being output.
● Input validation is very hard to do right. Examples where it goes wrong:
  ○ Encodings that the filter author forgets;
  ○ incomplete or bad regular expressions that do not catch everything;
  ○ if your valid input is essentially code, you would have to figure out what the code does;
  ○ broken libraries you depend on for (un)serialising / (un)marshalling;
  ○ false positives and too strict filters - the O'Donnells and Vähä-Sipilä's of the world always have fun.
● What would be the (academically speaking) correct way to do input encoding?
  ○ A state machine that is driven by specific inputs, with clearly defined state transitions that are caused by each symbol that is read.
  ○ This requires a formal grammar specification for the input. The challenge is that many real-life inputs are not easy to parse; they aren’t parseable by regular expressions or aren’t context-free grammars. Validating input where validation depends on something that was in the input previously (context-sensitive) potentially causes pretty hard validation challenges.
● Prepared statements are query language, and often database specific.
  ○ Example: INSERT INTO foo (?, ?, ?) values ($quux, $bar, $bletch)
● Output encoding in practice:
  ○ The idea is that attacker injection only works if the injected data is interpreted as code (i.e., directing execution) in the output context.
  ○ Example: A JavaScript function call “alert(1)” does not do anything in a text file, but opens a dialog if executed in a JavaScript program. The actual characters are the same; it is the output context.
  ○ Usually every output context has escaping rules where characters that would otherwise have special meaning will be transformed into something that only represents the character.
  ○ If every output context would escape or encode untrusted output with the rules applicable to that specific output context (e.g., HTML, JavaScript, SQL…), we do not need to do input validation.
Why output encoding might be a better strategy than input validation?
- When doing input validation, you do not yet know in which contexts the data is going to be output in the future.
- In output validation, the developer knows exactly where the untrusted data is being used, so specific escaping or encoding can be applied.
- There is less probability for data corruption due to poorly implemented, pre-emptive escaping or filtering.

What sort of attacks does output encoding not help?
- Those cases where the attacker wants to control the validation logic itself. Bad data could cause something to be evaluated incorrectly, thus passing a validity check. (E.g., bypass digital signature check through malformed input - particularly pressing issue when the input is complex and open to interpretation.)

Input TAINTING is a method that can be used to track untrusted input through the system.
- Mental picture: the use of a “contrast agent” in medical imaging
- The term “tainting” comes from Perl.
- Untrusted (i.e., potential attacker) input is marked as “tainted”, and the system ensures that tainted data cannot be output in any sensitive output context before being untainted (i.e., escaped or encoded).
- Some frameworks offer good support for this, e.g., Django (a Python web framework).

To summarise, some design and implementation paradigms for inputs:
- Aiming for regular and context-free inputs if possible, and implementing parsers for them using a parser generator (look at Bison and Yacc).
- Avoiding too expressive languages to convey data, if simple ones suffice. (Example: If all you need to provide can be expressed in a CSV, why use XML?)
- Not trying to guess what the input meant. Strict input validation and silent discard are often good alternatives, although compatibility issues may be in conflict with this principle.
Part 2: Privilege elevation on the web

NOTE: Cross-Site Scripting, discussed below, is one of the topics of the weekly exercise. It will not be demoed during the lecture, but if requested, we will demonstrate it during the weekly “lab hours”.

- Recap of session 1: Data is being run as code in the process context. In the demo and exercises last week, we were working in an environment where the lowest-level execution environment - the processor itself - is tricked into executing attacker’s code.
- The same idea can be pulled off on higher level languages. In these situations, it is typically an interpreter which is tricked into executing code that has been INJECTED into the original program. The details vary by language and environment.
- Since the advent of the World Wide Web, so-called “web applications” have become an extremely widely deployed set of programs. Originally, most of the logic was on the server end, and static web pages written in HTML were passed back and forth.

DISCUSS: How does HTTP work? How do cookies work? (This is supposed to be prerequisite knowledge, but it is useful to go through to ensure it is clear for everyone.)

DISCUSS: Which execution environments or languages exist in the web application space? Where are they executed? Where does the code that they execute, come from? (We are interested in, for example, JavaScript and SQL; both of these should be raised in this discussion.)

- Browsers are complex execution environments that execute JavaScript, which manipulates the document object model, which gets rendered based on HTML and CSS rules.
- Modern web applications no more pass static pages to and from the server, but instead pass (for example) JavaScript objects (JSON) back and forth, and alter the execution flow based on the content of those objects. What is interesting from security perspective - if you think about injection pathways - is that the data objects are actually executable code in their own right, or they can be evaluated as code.
- With mobile applications, we are also seeing more and more applications that are actually written in HTML and JavaScript, often using client-side programming frameworks and libraries. An example is Cordova (PhoneGap). These frameworks bridge the “native” mobile phone functionality with “web” applications.
- SQL is also a real programming language in its own right.
- Also, servers can be sometimes made to execute attacker-supplied code in their respective execution environment. For example, sometimes servers execute shell commands, or evaluate JavaScript objects they receive from the client.
One thing that protects the server side from a complete failure is that web servers usually run programs written in a higher level language, such as Python or PHP, where things like buffer overflows are less of a problem. Of course, these languages have their own failure modes and injection vectors.

Some ways of getting attacker’s code to run in a web application context. We will look at two: Cross-Site Scripting and SQL injection as examples.

Cross-Site Scripting (XSS)

- Browsers’ security model is Same-Origin Policy. The “origin” is a triplet of the URI scheme, domain, and port.
- Scripts from a specific origin can only access and manipulate data, and connect to, resources from the same origin.
- An XSS attack means that the attacker can inject code into a document which the browser thinks came from that specific origin, and thus can get access to something the attacker shouldn’t be able to access.
- XSS attacks come in two major flavours:
  - A reflected XSS attack sends the attack code in a request, and gets it back in a response. Because the code now “came” from the specific origin, it has elevated its privileges to act as if it really was intended to come from the server.
  - A stored XSS attack is like a reflected attack, but the attack code gets somehow stored in a database or some other place, and it gets played back without it being in the currently processed request.

In this regard, this is a PRIVILEGE ESCALATION attack, where privileges are given by the script origin. (Some might think this kind of shoehorns the attack into this class of issues, but I don’t think it’s that far fetched.)
DISCUSS: Why is a stored XSS attack worse? What kind of setup would a stored XSS attack be used for profit?

○ What an attacker can typically do with an XSS attack?
  ■ Manipulate the web page, and access anything the user types on the web page.
  ■ Steal cookies.
  ■ With hybrid JavaScript/native frameworks, potentially access native programming interfaces.

○ There are new web technologies such as Cross-Document Messaging and Cross-Origin Resource Sharing (CORS) which have an effect on the same-origin policy. We will not go into these in detail, but just to let you know that this area is under significant changes, and if you want to stay relevant in modern web application security, you need to read up on those.

● SQL injection
  ○ SQL is still the most prevalent back-end database query language. The rise of so-called NoSQL databases has eroded the popularity a bit.
  ○ A very typical SQL injection happens when attacker-supplied query terms are directly concatenated into an SQL query. They get executed as a part of the SQL query under the database user’s (application’s) privileges, and in this way this can also be construed as a privilege escalation.
  ○ What can an attacker typically do with an SQL injection:
    ■ Add, copy, or delete data.
    ■ Dump contents (many of the “data breaches” reported in the press have an SQL injection at their core).
    ■ New SQL users could perhaps be created, if the permissions are right.
    ■ If the SQL server (and the user on which the injection code is executed) provides some more esoteric services, such as access to the underlying OS environment, those might get used.

Part 3: Web session attacks

NOTE: Cross-Site Request Forgery, discussed below, is one of the topics of the weekly exercise. It will not be demoed during the lecture, but if requested, we will demonstrate it during the weekly “lab hours”.

● Web applications are stateless.
● In order to tie separate web page requests together, the server must have some information about the client. Usually this is done through creating a session, identified by a session identifier, and the client provides this session identifier to the server in every request.
● This is often implemented as a session cookie, which, as we remember, is sent automatically every time a request is made to that domain.
● This opens up various ways of attacking the user’s session.
● Cross-Site Request Forgery (CSRF, pronounced “cee-surf”)
- Another website directs a browser to target site
- The browser happens to be “in session” (e.g., logged in) to the target site
- The target site believes that this was an authentic user-initiated request and processes it
  - Mitigating these risks is a pain if you need to do this manually. Modern, good web frameworks provide protection against this type of issues (as well as good session control) out of the box.
  - Therefore, it is important to keep these issues in mind when doing technology selection for web application development.
  - Again, a new development on the web are Web Sockets, which are something we do not go into here, and you need to learn about if you want to do modern web app security testing in the future.

**Part 4: Web security testing with attack proxies**

NOTE: The weekly exercise involves the use of attack proxies. We will do a high-level introduction to them during the lecture to get you up to speed. If you need more face-to-face coaching, that is available during the “lab hours”.

- Typical exploratory web security testing utilises an attack proxy.
- An attack proxy sits between the client and the server, and allows the tester to modify server requests and responses.
- OWASP has a tool called Zaproxy (Zed Attack Proxy); there is a commercial tool with a lot of market share called the Burp Suite.

DEMO: We show both Burp Suite and Zap in practice, how they look like, and what concepts you need to understand about them.

Licensed under Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License (see https://creativecommons.org/licenses/by-nc-sa/4.0/).

Attribution: Antti Vähä-Sipilä, avs@iki.fi. Thanks to Sini Ruohomaa and Prof. N. Asokan.