CS412 Software Security

Web Security

Mathias Payer

EPFL, Spring 2019

Web (and internet) security: two views

- Just a long running network service
- Complex application with multiple layers and components.

Web (and internet) security: two views

- Just a long running network service
- Complex application with multiple layers and components.

Start with the software security view, then dive into multi-layered aspects.

Daemons / Services / Servers

> *A daemon is a long running service that serves outside requests. A web server, a mail server, or a DNS server are examples of daemons.*

What makes daemons prone to attacks?

Daemons / Services / Servers

> *A daemon is a long running service that serves outside requests. A web server, a mail server, or a DNS server are examples of daemons.*

What makes daemons prone to attacks?

- Daemons are long running
- Daemons are complex (multi-threaded, caching, broad functionalities)
- Daemons are exposed

Daemons are long running

- ASLR/stack canaries are probabilistic, single secret per process
- Heap layout influenced by concurrent allocations
- Information leaks become more dangerous

Daemons are complex

- Crashing threads are restarted: resilience/uptime versus security
- Large set of functionalities increases attack surface
- Shared secrets across users in single address space

Daemons are exposed

- Concurrent users must be serviced
- Outside connections are allowed
- Attackers can leverage many different IPs (what about rate limiting accounts?)

Daemon compartmentalization

- Break complexity into smaller compartments
- Develop “fault compartments”, can fail independently
- Goal: one component fails, others continue to function

Example: mail agent

- Mail agents need to do a plethora of tasks:
  - Send/receive data from the network
  - Manage a pool of received/unsent messages
  - Provide access to stored messages for each user
- Two approaches: sendmail and qmail
- Sendmail uses a typical Unix approach with a large monolithic server and is known for the high complexity and previous security vulnerabilities
- QMail uses a modern least privilege approach with a set of communicating processes.

QMail

- Separate modules run under separate user IDs (isolation)
- Each user ID has only limited access to a subset of the resources (least privilege)
- Only one very small component runs as `suid` root
- Only one very small component running as root

QMail

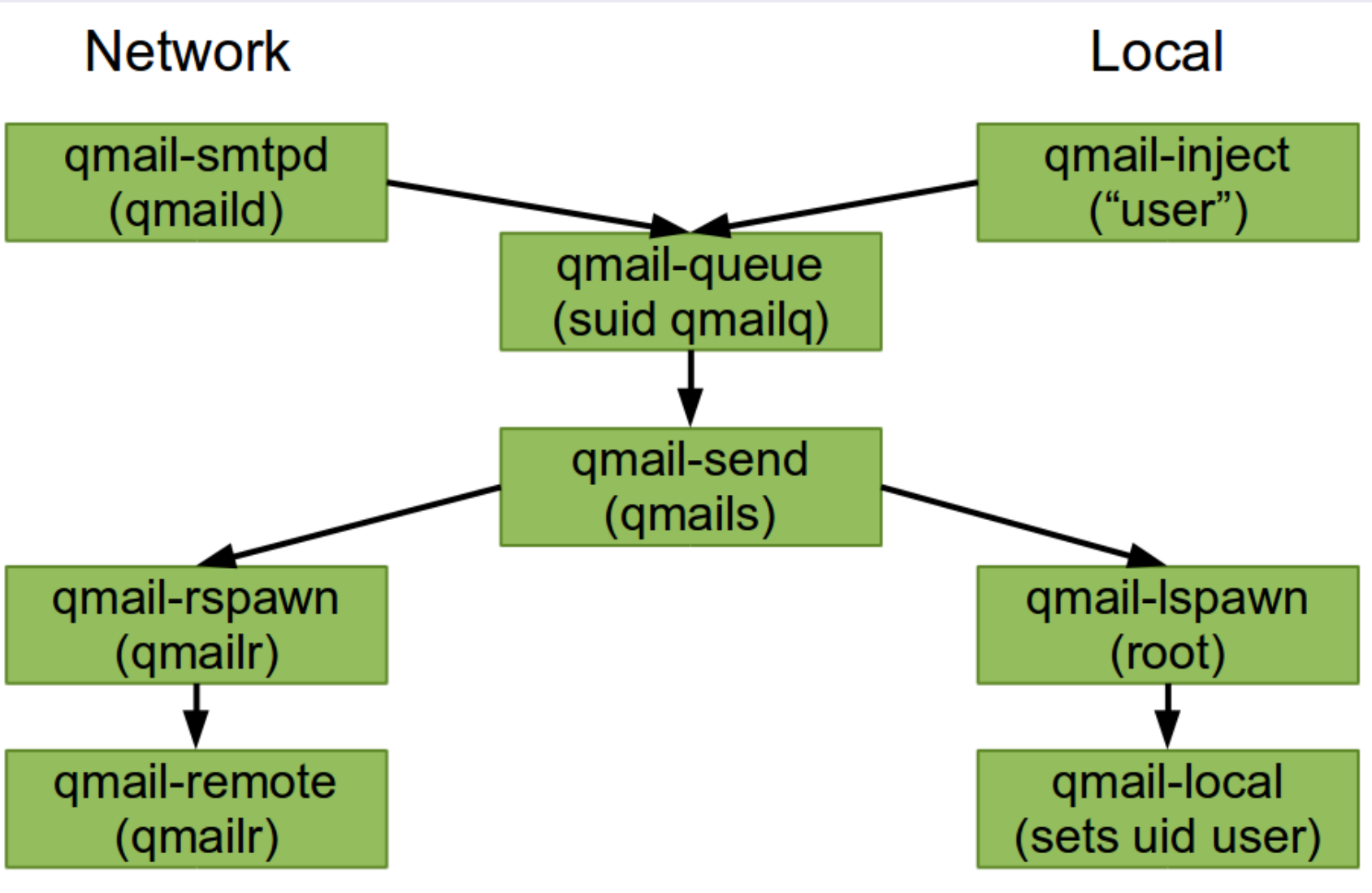

Figure 1:

QMail components

- *qmaild/user*: incoming email
- *suid qmaild*: split message into contents and headers, signal qmail-send
- *qmail-send*: send locally or remotely
- *qmail-lspawn*: root, spawns qmail-local with ID of user
- *qmail-local*: handles alias expansion, delivers locally, or signals qmail-queue if needed
- *qmail-remote*: sends remote message

OWASP Top 10

- OWASP: Open Web Application Security Project
- Collects information about vulnerabilities and attack vectors
- Releases top 10 of vulnerabilities every couple of years
- Most recent: OWASP Top 10, 2017

OWASP Top 10 (2017)

- (Code) Injection
- Broken Authentication
- Sensitive data exposure
- XML External Entities (XXE)
- Broken Access control
- Security misconfigurations
- Cross Site Scripting (XSS)
- Insecure Deserialization
- Using Components with known vulnerabilities
- Insufficient logging and monitoring

OWASP: What changed 2013 to 2017?

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Figure 2:

Top 1: Code Injection

> *Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization*

Code is not restricted to executable instructions. Web applications execute different kinds of code, many of which could be under the control of an attacker. Prime examples:

- Command injection
- SQL injection

Command injection

- Unix philosophy: leverage simple tools to achieve complex results
- Data is passed to scripts or programs as parameters
- Often the constrained communication channel will contain both code and data (e.g., the query command and the query arguments)
  - While functionality is tested, the security guarantees are often not
  - Vetting and escaping arguments correctly is challenging

Example: web-based command injection

- Dynamic web pages execute code on the server
- This allows the web server to add content from other sources (e.g., databases) and provide rich interfaces back to the user
- Build and combine complex parts dynamically and send the final result to the user (e.g., a content management system that loads contents from the database, intersects it with the site template, adds navigation modules and other third party modules)

Example: web-based command injection

```
<html><head><title>Display a file</title></head>
<body>
<? echo system("cat ".$_GET['file']); ?>
</body></html>
```

Example: web-based command injection

```
<html><head><title>Display a file</title></head>
<body>
<? echo system("cat ".$_GET['file']); ?>
</body></html>
```

- There is no separation of code and data that is passed through the channel
- `display.php?file=info.txt\%3bcat\%20\%2fetc\%2fpasswd`
- `;` allows chaining of individual bash commands
- `system` is a powerful command that executes full shell scripts

Command injection mitigation

- Can we just block ;?

Command injection mitigation

- Can we just block ;?
- Blacklisting is not a good solution, attack space may be infinite
  - What about using a pipe?
  - What about using a backtick?
  - What about other commands (cat instead of rm)
  - Even the shell has many builtin commands

Mitigation through validation

- Ensure that the filename matches a set of allowed filenames
- Non-alphanumeric characters are needed to execute commands
- Fix both directory and set of allowed files
- Disallow special characters in the file name

Mitigation through escaping

- Escape parameters so that interpreter can distinguish between data (channel) and control (channel)
- Escaped form: `system("cat 'file.txt')`
- How do you write such an escape function?

Mitigation through escaping

- Escape parameters so that interpreter can distinguish between data (channel) and control (channel)
- Escaped form: `system("cat 'file.txt')`
- How do you write such an escape function?

You don't – there's a huge potential for error. Use built-in ones.
Each language has its own flavours of escape functions.

Mitigation through reduction of privileges

- The system command is immensely powerful as it launches a new shell interpreter
- Fall down to simplest possible API: open the file yourself and read it into a buffer or, if you *must* execute a command, launch it directly and not through the shell

Generalized injection attacks

- What enables injection attacks?
- Both code and data share the same channel.
- In the system example above, cat and file are specified as part of the same "shell script" where ; starts a new command
- In code injection the data on the stack and the executed code share the same channel (as do code pointers)

Example: SQL injection

```
$sql = "SELECT * FROM users WHERE email='"
     . $_GET['email']
     . "' AND pass='" . $_GET['pwd']
     . ';"
```

- What is wrong with this query?

Example: SQL injection

```
$sql = "SELECT * FROM users WHERE email='"
     . $_GET['email']
     . "' AND pass='" . $_GET['pwd']
     . ';"
```

- What is wrong with this query?

- An attacker may inject ' to escape queries and inject commands.
- (Also, the password is not hashed but stored in plaintext.)
- SQL injection is, in spirit, the same attack as code injection or command injection.

SQL injection mitigation

- Same idea: validation, escaping, or reduction of privileges.
- Separate control and data channel: prepared SQL statements
  - Similar to printf, define "format" string and supply arguments

```
sql("SELECT * FROM users WHERE email=\$1 AND pwd=\$2", emai
```

How do you prevent code injection?

- Use a safe API (e.g., prepared statements in SQL or restricted commands)
- Validate input based on a whitelist, reject invalid input
- For any remaining dynamic query data, escape commands using available API (do not roll your own escape functions)

Top 2: Broken Authentication

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

- Multi-factor authentication to mitigate brute-force attacks or PW leaks
- No default credentials
- Develop a reasonable password policy (length, strength, checks)
- Limit or delay failed login attempts
- Log any login failures

Top 3: Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

- Be aware of what data types are handled (threat modeling)
- Only store what you need, nothing more
- Protect data at rest and in transit (e.g., through encryption)
- Only store password hashes

Top 4: XML External Entities (XXE)

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

- Code injection: make sure that you don't include external or untrusted resources (see top 1)
- Use simpler data formats such as JSON
- Avoid serializing sensitive data (see top 8)
- Disable XML External Entities and DTD processing in XML parsers

Top 5: Broken Access Control

Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

- Except for public resources deny by default
- Rely on central access control
- Log access control failures

Top 6: Security Misconfiguration

Security misconfiguration is a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, or error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

- Configuration documentation (what software is in use? In what configuration?)
- Keep your systems up to date! (see top 9)
- Use compartmentalization
- Test for configuration failures

Top 7: Cross-Site Scripting (XSS)

> *XSS allows an attacker to inject and execute JavaScript (or other content) in the context of another web page (e.g., malicious JavaScript code that is injected into the banking web page of a user to extract user name and password or to issue counterfeit transactions).*

Three kinds of XSS: persistent/stored, reflected, and client-side XSS.

Persistent XSS

- The attacker stores the attack data on the server itself
- A simple chat application allows users to store arbitrary text that is then displayed to other logged in users
- The attacker may send a message that contains `<script>alert('Mr. Evil here');</script>`
- Common use case: feedback forms, blog comments, or even product meta data (you don't have to see it to execute it)
- Bug is on the *server side*

Reflected XSS

- The attacker encodes the attack data in the link that is then sent to the user (e.g., through email or on a compromised site)
- A web interface may return your query as part of the results (i.e., "Your search for 'query' returned 23 results.")
- Requirement: user must click on attacker-controlled link
- Bug is on the *server side*

Client-side XSS

- Large applications contain lots of JS code. Code itself may contain vulnerabilities.
- JS may use URL parameters to process information (think AJAX/JSON requests to process data in the background)
- Attacker must make user follow the compromised link but, compared to reflected XSS, the server does not embed the JavaScript code into the page through server side processing but the user-side JavaScript parses the parameters and misses the attack
- The bug is on the *client side*, in the server-provided JS

Mitigating XSS

- Modern frameworks allow you to properly escape input. Each framework has its subtleties, be careful
- Leverage data-flow to detect where unfiltered input exists

Top 8: Insecure Deserialization

Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

- Integrity check (MAC) messages
- Enforce type checks during deserialization (e.g., flag an error if application receives an array or map instead of a string)
- Isolate parsing code to deprivileged process
- Log deserialization exceptions

Top 9: Using Components with known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

- Remove unused dependencies
- Documentation of components and their dependencies (see top 6)
- Monitor for unmaintained libraries or components

Top 10: Insufficient Logging & Monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breaches are detected late (200+ days) and by external parties.

- Ensure errors are logged and log files are evaluated
- Audit trail for important transactions, clear responsibilities

OWASP Summary

OWASP Top 10: (code) injection, broken authentication, sensitive data exposure, XML external entities, broken access control, security misconfiguration, cross-site scripting, insecure deserialziation, using components with known vulnerabilities, insufficient logging and monitoring.

- Input parsing: (code) injection, XML external entities, cross-site scripting
- Authorization: broken authentication, sensitive data exposure, broken access control
- Configuration and documentation: security misconfiguration, using components with known vulnerabilities, insufficient logging and monitoring

Summary and conclusion

- Daemons are long running, complex, and exposed.
- Compartmentalization helps reduce attack surface
- Command/SQL injection is a powerful attack across many applications
- Separation of code and data is crucial
- OWASP Top 10: input parsing, authorization, and configuration/documentation