CS527 Software Security

Web Security

Mathias Payer

Purdue University, Spring 2018

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

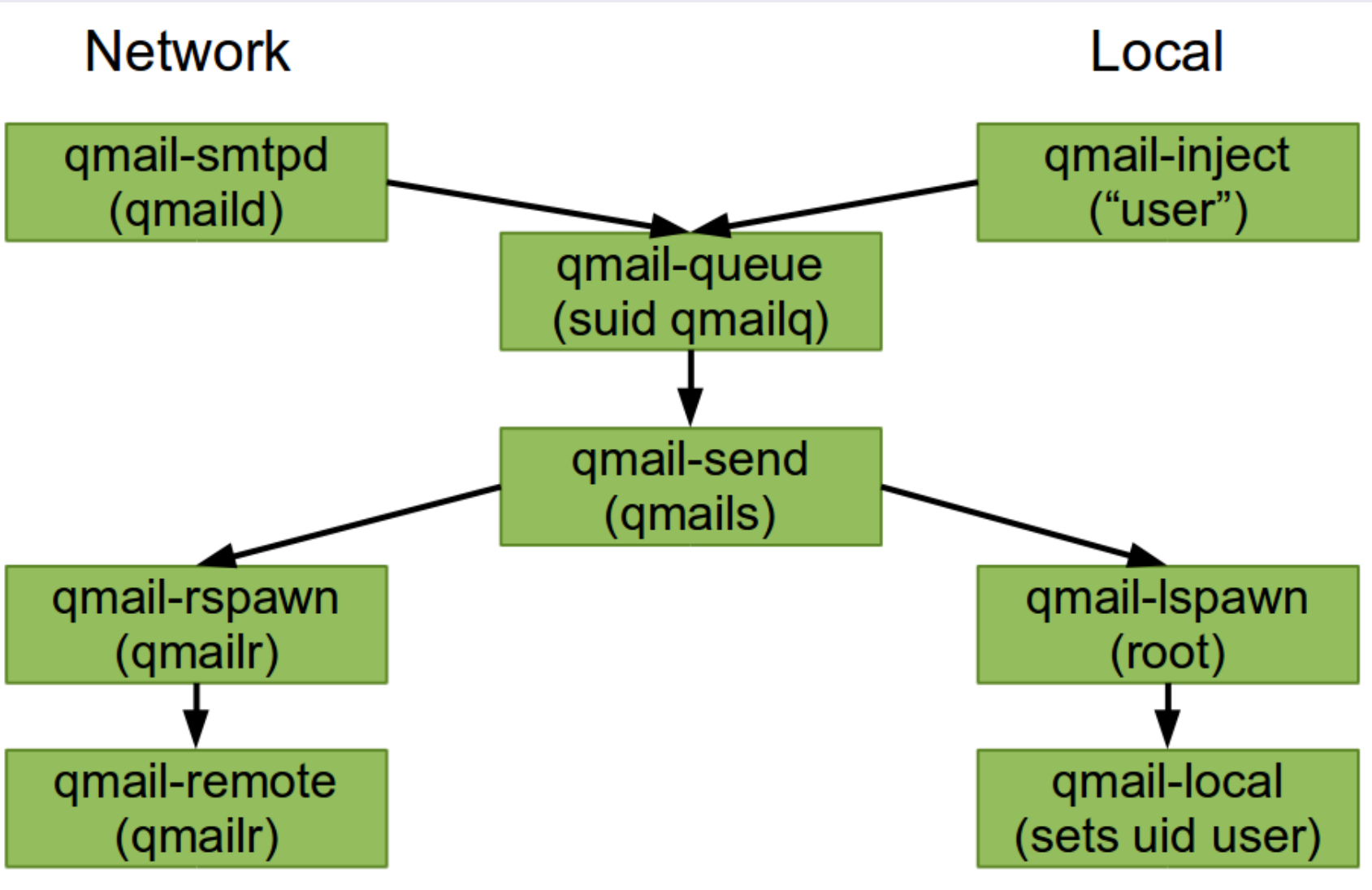

Figure 1:

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

Case study: daemon attack surface

- Command injection
- SQL injection
- Cross site scripting

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
```

Cross Site Scripting

> *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

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
- XSS allows execution of malicious JS code *through* a web application
- All three types of attacks share the common problem that code and data are not separated