Application Security (apsi)

Lecture at FHNW

Lecture 3, 2021

Arno Wagner, Michael Schläpfer, Rolf Wagner

<arno@wagner.name>, <{michael.schlaepfer,rolf-wagner}@fort-it.ch>

Agenda

- Test-before-use, Input Validation
- Input Normalization
- Privilege models in Linux (Unix): file access, running processes
- Chroot(2)
- Mandatory Access Control (MAC): SELinux, AppArmor
- Architecture-Pattern: Privilege Separation

Secure Coding Principle: Test-before-use

This is a fundamental secure coding principle

- Always test <u>all</u> assumptions about data before using it in any way
  → This makes assumptions explicit and <u>documents them</u>
- Code only has to work for the tested propertie
  → This simplifies the code and helps with KISS
- Examples
  - Length of strings and buffers
  - Allowed characters in strings
  - Numerical ranges
  - State of sockets or files
  - ...

Input Validation

Purpose: Establish properties of input data first
→ Instance of the "test-before-use" principle

- Later processing stages can then depend on the tested properties
  → This makes coding and error handling much easier.

What needs to be validated?

- *Every* property of input data the program depends on!
  → You may <u>only</u> assume input data properties you have validated first!
  (Insecure software is almost always caused by invalid assumptions…)
- Examples:
  - Input length (min, max) → Buffer overflow. "Too short" can also be a problem.
  - Input character set, escaping or not, special codes (for example \0)
  - Other formats, e.g. "valid XML", "valid JSON", "valid HTML", etc.
  - Configuration files

Input Normalization

This is a preprocessing step, used to decrease variability of input data

The goal is simplification of further processing (again "KISS")

- Reduce a larger set of input formats to a smaller one or to a single format
- Simplifies following format-validations and processing
- May have input constraints as well, input validation before still needed

Examples:

- Decompress data or images for multiple compressors and (optionally) recompress with a specific one
- Recode all text-input to UTF-8
- Virtually print a document, use OCR on it to recover the text
  → This is currently the only really secure way to sanitize documents.

Unix Permission Model (simplified)

1) Everything is a file (includes sockets, directories, interfaces, etc.)
2) Every file has 3 permisson sets: user, group, other
3) Every file can be (r)readable, (w)ritable and e(x)ecutable in each group
   Note: For directories "x" is required for chdir-ing into them.
   "x" becomes "s" on executable files with suid/sgid bit set
4) A running process has a user and a group which determins its rights

- This is simple, but often enough. If not → Mandatory Access Control (MAC)
- "root" is special and can change all permissions
- A process running as root can "drop privileges" to the permissions of a regular user by calling setgid() and setuid(). (Done right, it cannot get back.)
- The suid/sgid-bit executes that file as the user/group it belongs to.
  → This allows ordinary users to execute programs as root
  Example: /bin/su: `-rwsr-xr-x 1 root root 36816 May 25  2012 /bin/su`

Chroot

Sets the apparent root-directory ("/") for a process

- Isolation technique
- If done right, process cannot access files outside
- Usually used before a "privilege-drop"
- Can be used to "trap" a running process in a subdirectory

Limitation:

- Root can beak out of a chroot situation

Note: There is both a C call chroot(2) and a shell-call chroot(8)

Reference:
- http://www.unixwiz.net/techtips/chroot-practices.html

Mandatory Access Control (MAC)

Fine-grained access control enforced by the OS kernel

- Allows very specific permissions
- Allows restricting "root" (usually needs reboot to change)
- Knows OS objects are more than just files (sockets, directories, etc.)
- Has a concept of groups, allows control over entering and leaving groups
- Cannot be switched off at run-time (unless specifically allowed)
- Allows a "fully locked down" system, even against root.

But:

- Difficult to configure
- Can cause hard to debug run-time problems
- May hinder normal system administration

MAC Example: SELinux

- Initial release: 1998, created by the NSA (this should not be a problem)
- Does labelling for everything in the file-sytem
  → can take a while on first start
  Needs filesystem that supports security-labels
- 3 Modes: enforcing, permissive, disabled
- Can restrict what root can do, can prevent root login
- Complex to configure and maintain
- Very fine-grained control

Reference: https://en.wikipedia.org/wiki/Security-Enhanced_Linux

MAC Example: AppArmor

- Initial release: 1998
- Works with file-paths (i.e. not files directly)
- Simple (relatively to SELinux) to configure
- Does not need a file-system with security labels
- Can be switched-off on a per-path level
- Less fine control than SELinux
- Less structured than SELinux

Reference: https://en.wikipedia.org/wiki/AppArmor

Secure Coding Principle: Defense in Depth

Architecture and coding principle

Idea: Have several security mechnisms so that one is enough to still be secure if the others have been circumvented by an attacker.

- Example: Input validation and least privilege
- Example: Password and smartcard

Rationale:

- Gives resilience if one mechanism fails (and mechanisms <u>will</u> fail...)
- Gives resilience against bugs in the implementation of mechanisms
- Can help in attack-detection
- Allows individually less-complicated protecion mechanisms (KISS)
  → Complex protection mechnisms can be a problem! Example: IPSec
- Makes attacks more expensive → Discourages attackers

Privilege Separation

- Consequence of the Principles of "Least privilege" and "Defense in depth"
  Idea: Run code only with the minimal privileges needed

But:

- Code is complex and different parts need different privileges

Hence:

- Sepearate code into modules, run each one with minimal privileges needed
- In particular seperate things like input normalization, network and system access, user interaction, etc.
  → Modules later in the data-flow are not directly exposed
  → If data is incorrect later, refuse processing

This technique has allowed some very exposed server software to survive well in a very hostile Internet

Overall Idea

Two main approaches (will be combined):

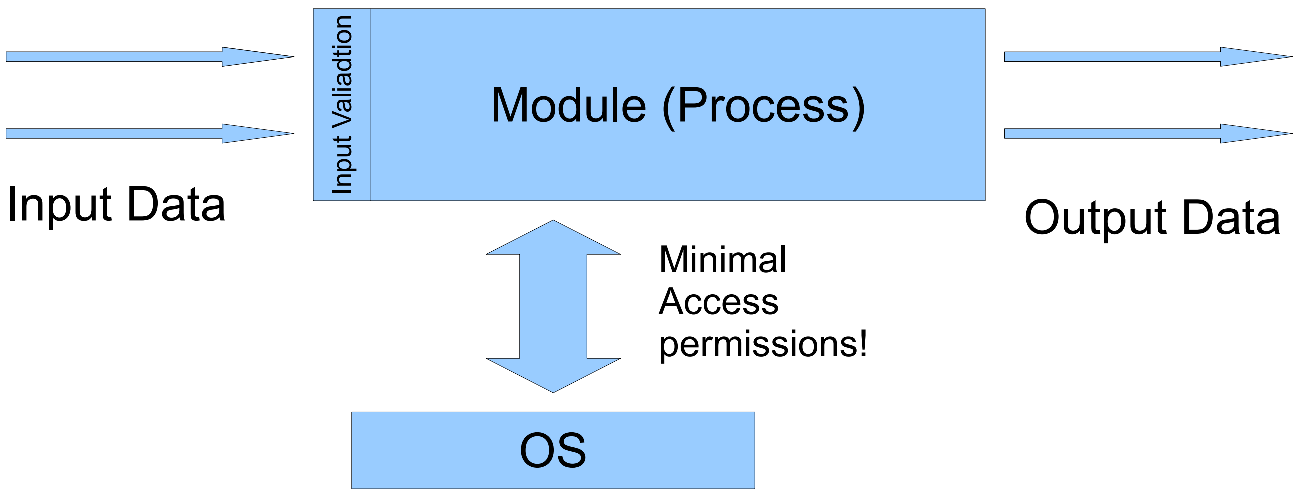
1) Operations that require high privilege (root) are done before any client interaction, then drop privileges before procesing data
   - Example: Allocating a network socket (ports below 1024 require root-rights)
   - Example: Accept a connection but then hand it to a non-privileged child
   - Example: Allocating non-swappable memory

2) Divide processing into modules along the data-path
   - Input validation first and again in each module
   - Normalization next
   - Actual processing next

Communication between elements: Files and usual Unix IPC like local sockets, pipelines, signals, ...

Form of the Modules

- A module is encapsulated into one or more processes (or machine, container, etc. for larger security needs)
  → Run each module with the minimal access needed
  → If higher privileges needed for some steps, do them <u>before</u> processing input data and then drop privileges
- Do full input validation in each module!
- If module seems to require 2 sets of privileges for data processing, split it!



Principles for Privilege Separation

- Always do a full (!) input validation on any internal communication channel
- Keep internal communication as simple as possible (see "KISS")
  -> This makes input validation simpler and reduces vulnerabilities in it
- If input validation fails at an internal channel, if possible log the issue and optionally find the sending component to be compromised. (Application-internel intrusion detection.) Consider preparing contermeasures.
- Allways make sure compromising another module from a compromised one is as hard as possible
- Do not expose the internal communication
- Treat configuration files, data files, etc. as communication channels
- Select the right level of separation: Hiogh security requirements need separation ins small components, low security requirements can use larger, less restricted components.

Suitable and Unsuitable Communication Mechanisms

Suitable for communication between privilege-separated modules:

- Anonynmous and named pipes, local sockets ("Unix Domain Sockets")
- Signals
- Files in the Filesystem (with proper restrictions by permissions)

Note: If forking children: Before you do anything else:
1. Delete all file-descriptors from the parent that the child does not need
2. Erase all memory that contains any secrets of the parent

Unsuitable mechanisms:

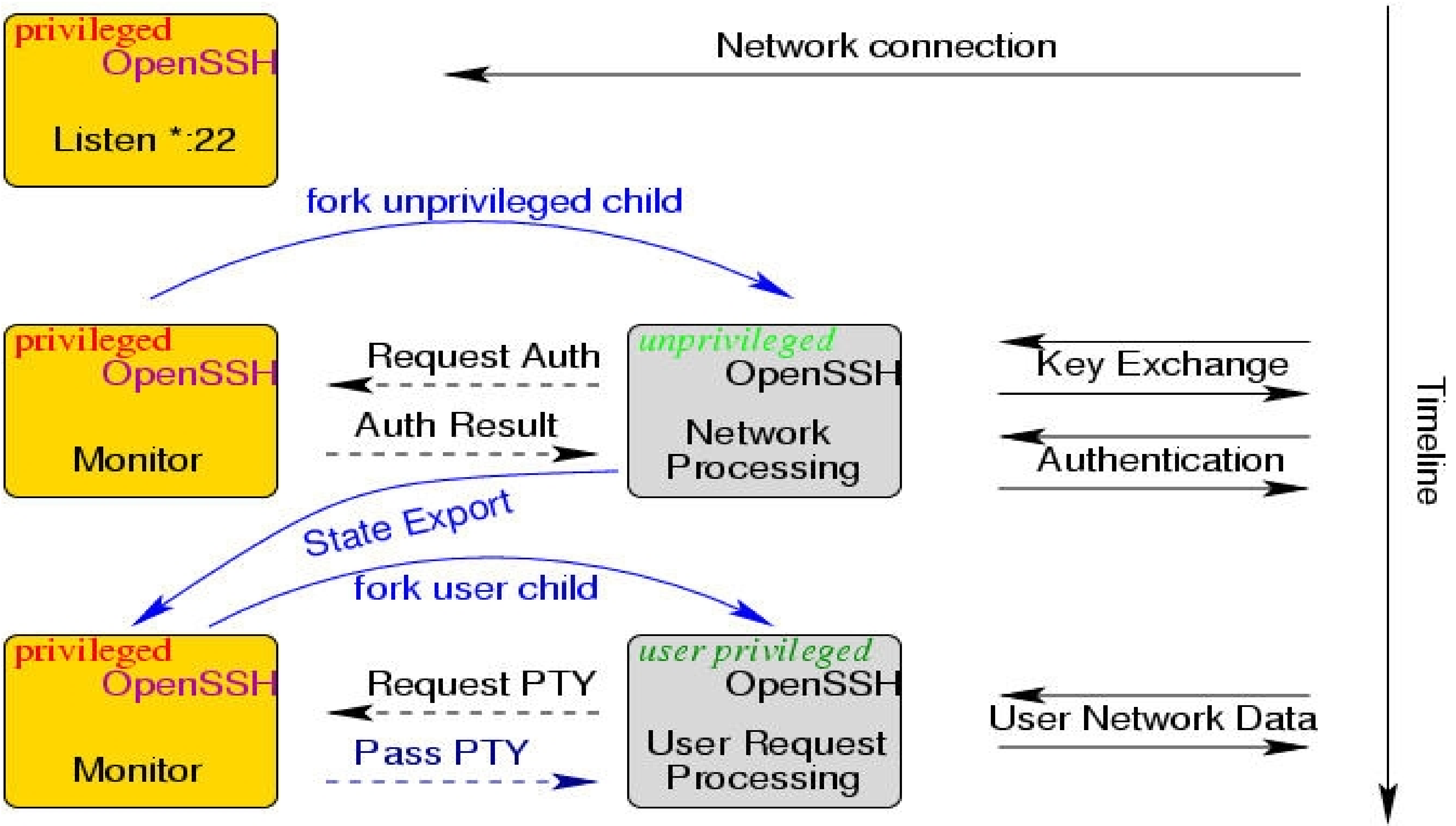
- RPC (Remote Procedure Calls): Have a very bad security track record
- Shared memory: A buffer overflow can propagate…
- Network sockets: You expose internal communication to the network!
  If using 127.0.0.1, you still expose internal communication to other rpocesses!
- Sending serialized objects: You may end up sending compromised objects!

Example: OpenSSH

This is the OpenBSD Secure Shell Server

- Initial release 1999
- The standard way for secure shell login to a UNIX machine
- Obviously an attractive target
- Must be Internet-reachable in most cases
- Must protect private keys used for authentication
- Details: http://www.citi.umich.edu/u/provos/ssh/privsep.html

OpenSSH: Privilege Separation Approach



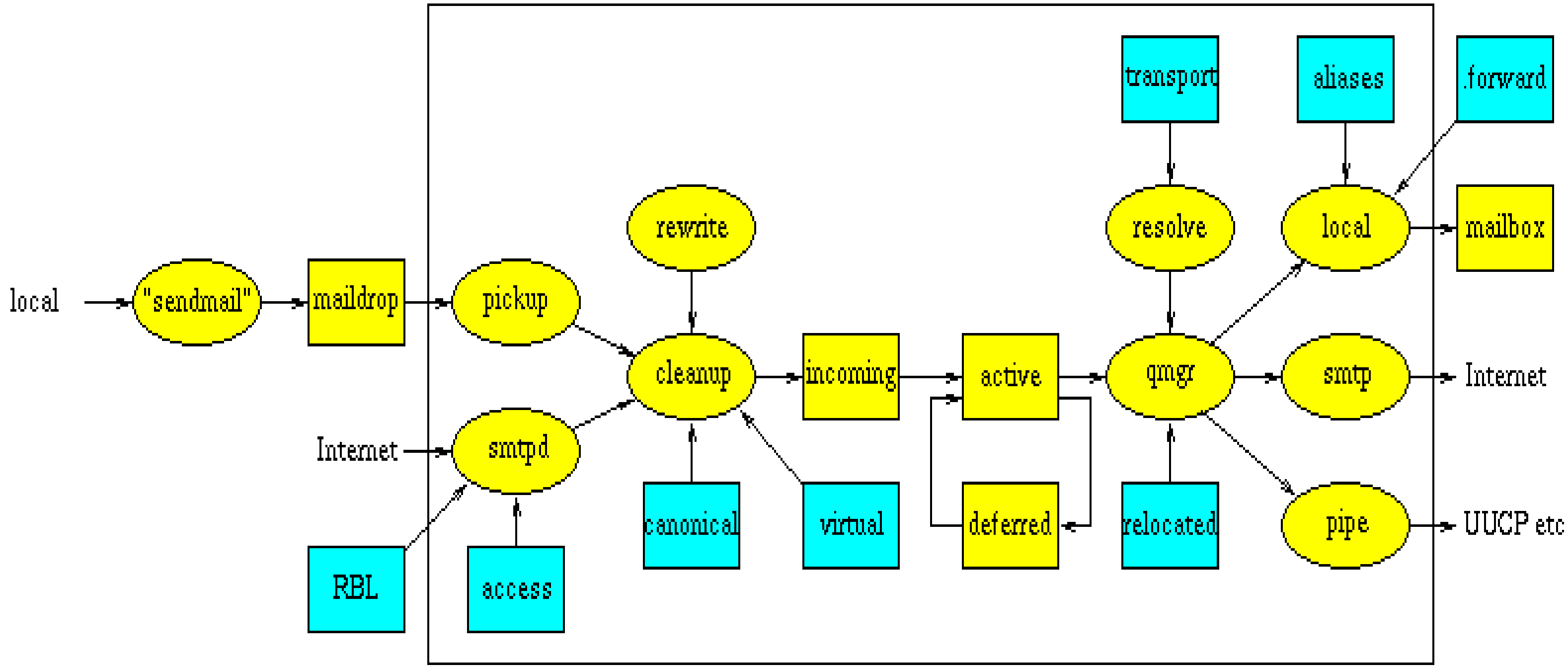
Example: Postfix MTA (Email Server)

The Postfix secure Email Server

- Initial release 1998
- Large, complex Email handling server (smtpd)
- Must be Internet-Reachable without any authentication or limit as to source
  Possible exception: SPAM-sources
- Must open connections to arbitrary other MTAs
- "In April 2016 in a study performed by E-Soft, Inc., approximately 32% of the publicly reachable mail-servers on the Internet ran Postfix."
- Email is often a critical communication channel

Reference: http://www.postfix.org/documentation.html

Postfix System View



Postfix: Example Data-Flow

Incomming email → local user

smtpd(8) -> cleanup(8) -> incoming -> qmgr(8) -> local(8)

- **smtpd**: Accepts email from the Internet.
  Filehandles/sockets and pipeline for giving mail to cleanup is inherited.
  Process has no permissions at all.
- **cleanup**: Validates and normalizes email-format.
  Is chroot'ed on the email-queue-directory, to which it can only write, not read.
  Is allowed to send signals to **qmgr**.
- **qmgr**: Decides where to send (local, send over the net, pipe to command).
  Only reads email header, not content or attachments.
  Is chroot-ed on several mail-queue directories.
  Allowed to signal the delivery demons **local**, **smtp**, **pipe**
- **local**: Does local delivery. Runs as root.
  Gets target-user file-name in queue-directory via pipe.
  Moves file to user's mailnox, but never reads it.