Takeaways from DJB’s approach to secure software development

Vítor Francisco Fonte, vff@di.uminho.pt, University of Minho, 2024

Qmail

- A mail transfer agent (MTA), sends and receives local and remote mail
- Development started late 1995, final public release in 1998 (v1.03), by Daniel J. Bernstein (aka djb)
    - Replacement for Sendmail (prevalent at the time)
- Modular security-aware architecture composed of mutually untrusting components
    - e.g. the SMTP listener component runs with different credentials from the queue manager or the SMTP sender
- Backed with a security guarantee
    - 500 USD prize form the author for confirmed each security bug (later upgraded to 1,000 USD)
    - More info available: https://cr.yp.to/qmail/guarantee.html

Qmail vs Sendmail

- Fewer lines of code, minimal and uniform feature set
- Service decomposed in multiple processes (vs. monolithic service)
- Processes run with different uids and are resource restricted (vs. root user)
- Processes don’t trust each other and always validate their inputs first
- Impressive security record vs. hundreds of security vulnerabilities
  - vulnerabilities found for the first (and last) time in 2020, related to integer representation with impact resource usage and exploitable as remote program execution under local users (but not root!)

A future with fewer and less harmful bugs

- A number of takeaways from DJB's approach to software security can be applied to the development of security-aware software, in general:

  1. Eliminate bugs / write bug-free code and practice efficient coding

  2. Eliminate code / follow a simple, clean modular architecture

  3. Reduce the trusted code base

- We also add a few remarks to DJB's insights

  - The qmail security guarantee, https://cr.yp.to/qmail/guarantee.html

  - Daniel J. Bernstein, "Some thoughts on security after ten years of qmail 1.0", Proceedings of the 2007 ACM workshop on Computer security architecture, https://cr.yp.to/qmail/qmailsec-20071101.pdf

Eliminate bugs

- Use modularity and encapsulation
- Always avoid global variables
- Always check against the bounds of arrays before accessing it (e.g. buffer overruns)
- Always mind limitations of quantity representation in programming languages (e.g. integer overruns)
- Always check resource availability
- Always free resources as soon as they are not needed
- Always write a test for every feature of the code
- Whenever possible, use better software development toolchains and processes
  - e.g. programming languages with automatic extension of arrays
- Always follow defensive programming practices
- Code must be easy to read
- Code must adhere to a uniform coding standard
- Explicit over implicit

Eliminate code

- Code must be as small as possible
- Implement a minimal feature-set
- Refactor code as much as possible
- Take advantage of the OS mechanisms and abstractions
    - e.g. access control, processes, interprocess communication, file system, and service infrastructure

Reduce the trusted code base

- Do as little as possible in setuid programs
- Do as little as possible as root
- Move separate functions into mutually untrusting programs
  - programs run with different uids
  - programs don’t trust their inputs (validate first, act later)
  - programs run with minimal privileges and as resource restricted as possible (see next slide)
  - djb suggests structuring the software as transformation functions (aka UNIX filters)
    - e.g. parsing function (parsing is always error-prone!)

Simple recipe to restrict process execution

- Prohibit new files, new sockets, etc., by setting the current and maximum RLIMIT_NOFILE limits to 0.
- Prohibit filesystem access: chdir and chroot to an empty directory.
- Choose a uid dedicated to this process ID. This can be as simple as adding the process ID to a base uid, as long as other system-administration tools stay away from the same uid range.
- Ensure that nothing is running under the uid: fork a child to run setuid(targetuid), kill(-1,SIGKILL), and _exit(0), and then check that the child exited normally.
- Prohibit kill(), ptrace(), etc., by setting gid and uid to the target uid.
- Prohibit fork(), by setting the current and maximum RLIMIT_NPROC limits to 0.
- Set the desired limits on memory allocation and other resource allocation.
- Run the rest of the program.