Security Measures in OpenSSH

Damien Miller

ダミアン　ミーラー

djm@openbsd.org

Introduction

- Describe the security measures in OpenSSH
  - What they are
  - How we implemented them
  - How well they work
- Why?
  - OpenSSH is an important and widely used network application
  - To convince you to use these techniques in your software

"Only failure makes us experts"

OpenSSH overview

- Project started in September 1999
  - Portability project started one month later
  - Killed telnet and rsh within two years (except for some router manufacturers)
- Most popular SSH implementation (over 87% of servers)
- Written for Unix-like operating systems
- Based on legacy codebase
  - Incremental approach to development

Our darker moments…

- Critical security problems (remote exploit):
  - deattack.c integer overflow (Zalewski, 2001)
  - channels.c off-by-one (Pol, 2002)
  - Challenge-response input check bug (Dowd, 2002)
  - buffer.c integer overflow (Solar Designer, 2003)
  - Incorrect PAM authentication check (OUSPG, 2003)
- More lesser bugs (we take a paranoid view and announce everything - exploitable or not)
- But also…
  - Zlib heap corruption (Cox, et al., 2002)
  - OpenSSL ASN.1 bugs (NISCC and Henson, 2003)
  - Zlib inftrees.c overflow (Ormandy, 2005)

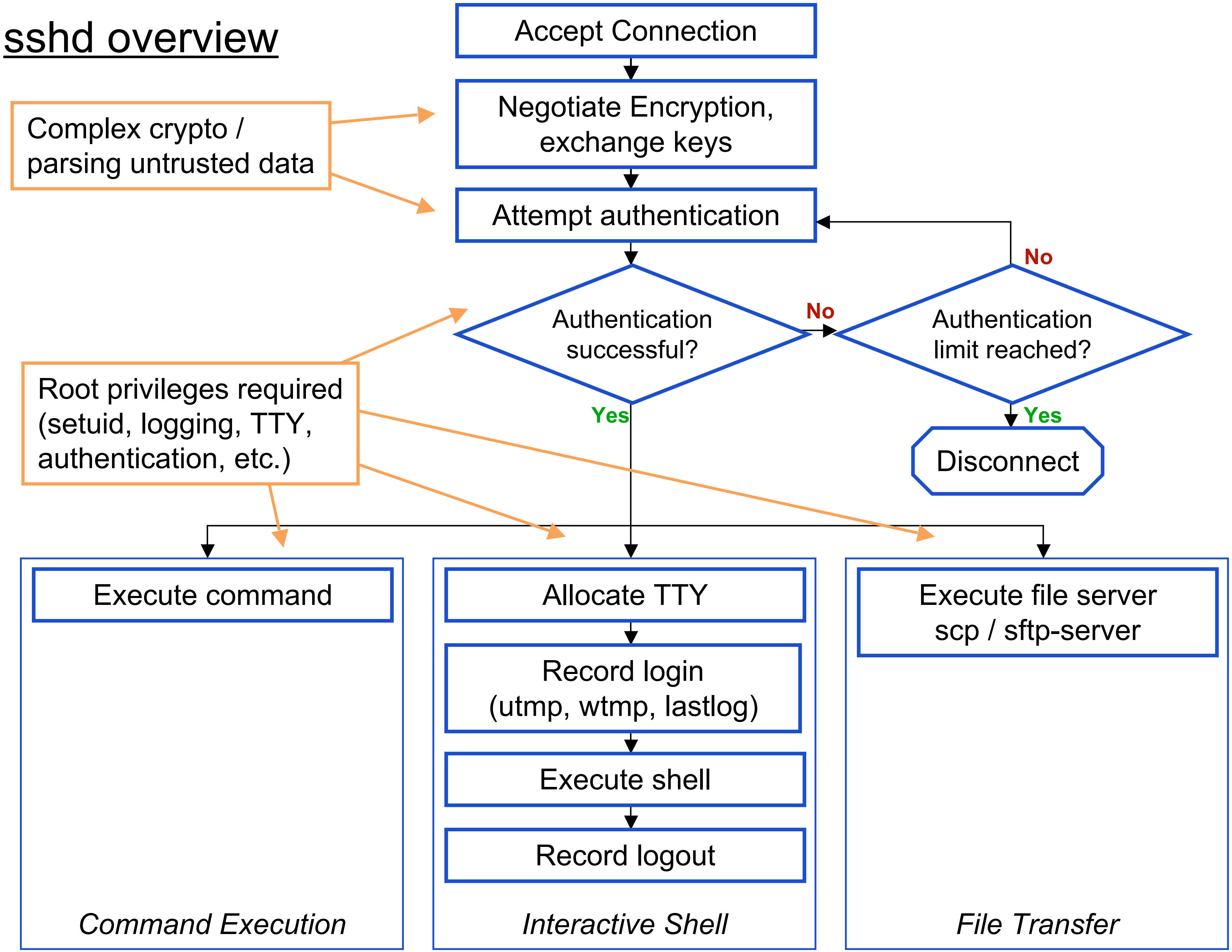
Attack surface[1]

- Amount of application code is exposed to attack
  - Scaled up for code that is exposed to anonymous (unauthenticated) attackers
  - Scaled up for code that runs with privilege
- The less the better!
- Corresponds to Saltzer and Schroeder's "*Simplicity of Mechanism*" and "*Least Privilege*" design principles[2]
- Good qualitative measure of system "attackability" (quantitative variants exist)

[1] M. Howard, "*Fending Off Future Attacks by Reducing Attack Surface*", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure02132003.asp, 2003

[2] J. H. Saltzer and M. D. Schroeder, "*The protection of information in computer systems*", pp. 1278-1308, Proceedings of the IEEE 63, number 9, September 1975

sshd overview



What can we do?

- Audit
- Add paranoia (defensive programming)
- Replace or modify unsafe APIs
- Replace complex and risky code with limited implementations
- Minimise / separate privilege
- Change the protocol
- Help OS-level security measures work better

Auditing

- OpenSSH has been repeatedly audited throughout its life
- Auditing does not mean *"find a bug and fix it"* - it means *"find a bug, and fix the class of problems its represents"*
  - If a developer makes a mistake, they are likely to have made it multiple times
- Bugs **will** slip through audits - most of the previously mentioned ones did.
- *Necessary, but not sufficient*

Paranoia / input sanitisation

- Input sanitisation is a necessity for all network applications
- Avoid passing untrusted data to system APIs (or any complex API) until it has passed basic format, consistency and sanity checks
- Constrain values to expected ranges
  - Integer overflows are a particular concern
  - Denial of service by allocating large amounts of memory
- Criticism: checks can bloat code
- Criticism: infeasible to catch every pathological case

Elimination of unsafe APIs

- Some APIs are difficult or impossible to use safely:
  - In 2007, the worst offenders are long gone
  - strcpy, strncpy ➔ strlcpy, etc. were done early
- Some are safe, but are simply painful to use:
  - strtoul() needs seven lines of support to robustly detect integer parsing errors[1]
  - Use strtonum()
- Some have subtle problems:
  - setuid() - may not permanently drop privileges on all platforms[2]
  - OpenSSH replaced with setresuid()

[1] Paul Janzen, *Examples section of OpenBSD strtol manual page,* 1999

[2] Hao Chen, David Wagner and Drew Dean, "*Setuid Demystified*", pp. 170-190, Proceedings of the 11th USENIX security symposium, 2002

Change the API

- Certain APIs lead to coding idioms than lend themselves to unsafe use
- Example: POSIX's use of -1 as an error indicator
  - Overloading of return value as both a quantity and error indicator encourages the mixing of signed and unsigned types, leading to integer overflows

```
size_t rlen = read(fd, tmpbuf, tmpbuf_len); /* (oops!) */
if (r < 0 || r > sizeof(buf))
  return -1;
memcpy(buf, tmpbuf, rlen);
```

  - Change the API – OpenSSH's *atomicio* read/write wrapper returns unsigned
- New code should not overload return value:
  - E.g. return quantity via size_t* argument

Change the API

- Dynamic array initialisation is frequently a source of integer overflows
  - malloc/realloc argument is almost always a product

```
struct blah *array = malloc(n * sizeof(*array));
/* later… */
array = realloc(++n * sizeof(*array));
```

- (n *sizeof(*array) > SIZE_T_MAX) -> wrap!
- Change the API: overflow checking allocators:

```
struct blah *array = xcalloc(n, sizeof(*array));
/* later… */
array = xrealloc(array, ++n, sizeof(*array));
```

  - Ensure that `(SIZE_T_MAX / nmemb) >= size`

Change the API

- Don't be constrained by an unsafe API
- Like auditing:
  - Treat the discovery of a bug as evidence that some wider may be wrong
  - Fix the underlying problem
- Criticism: inventing new APIs can make an application's code harder to read or learn
  - Choose sensible function names
- If we had implemented the xcalloc/xrealloc change sooner, we would have avoided at least one bug!

Replacement of complex code

- Very complex code can lurk beneath a simple function call
- Example: RSA and DSA signature validation
- Previously used OpenSSL RSA_verify and DSA_verify
- Called for public key authentication
  - I.e. 100% exposed to pre-auth attacker
- OpenSSL uses a full ASN.1 parser
  - ASN.1 is very complex and deeply scary
  - Nearly 300 lines of code, not including memory allocation, logging and the actual crypto
  - Has had remotely exploitable bugs

Replacement of complex code

- Replaced with minimal version that use fixed signature representations (no ASN.1)
  - Still use raw RSA/DSA cryptographic primitives
- Criticism: separate implementation does not benefit from ongoing improvements to mainstream version
  - So far, has not needed any maintenance
- This saved us from quite a few bugs:

  CVE-2003-0545, CVE-2003-0543, CVE-2003-0544, CVE-2003-0851, CVE-2006-2937, CVE-2006-2940, CVE-2006-4339 (Bleichenbacher e=3 RSA attack)

Privilege separation

- Very important design principle: applications should run with as little privilege as possible
- Example: Apache web server
  - Requires privilege to bind to low numbered ports, open log files, read SSL keys, etc.
  - Drop privilege before handling network data
- Result: a compromise gives an attacker access to a low privilege account
  - Can still locally escalate privilege
  - chroot/jail helps
- This model does not work for OpenSSH as it needs privilege throughout its life

Privilege separation

- Solution: privilege separation[1] - split the application:
  - *monitor* - handle actions that require privilege
  - *slave* - everything else (crypto, network traffic, etc.)
- The monitor should be as small (code-wise) as possible
  - Less code -> smaller attack surface, fewer bugs
- *slave* is always chrooted to `/var/empty`
  - Only access to system is via messages passed with *master*
  - Only escape is via kernel bugs

[1] Niels Provos, "*Preventing privilege escalation*", Technical report TR-02-2, University of Michigan, CITI, August 2002

Privilege separation

- For OpenSSH privilege separation (privsep), there are three different levels of privilege:
  - *monitor* -> always root
  - *slave* before user authentication -> run as dedicated user
  - *slave* after user authentication -> run as logged in user
- Note that a compromise of a post-auth slave does not gain the attacker any more privilege
- When first implemented, estimated privilege reduction was ~66% (measured in lines of code)

Privilege separation

- Splitting unprivileged code from privileged is insufficient:
  - Attacker compromises slave
  - Fakes messages to master, requests system access
- So the monitor must enforce constraints on what privileged actions that slave may request of it
  - Do not spawn subprocesses before authentication
  - Do not allow unlimited authentication attempts
  - Some requests will occur only once in a normal protocol flow
- OpenSSH's monitor is structured as a state machine
  - Bonus: second, independent layer of authentication checks serves as safeguard against logic errors

Privilege separation

- Next problem: a SSH connection requires a significant amount of state
  - Crypto keys and initialisation vectors, input/output buffers
  - Compression (zlib) state
- When authentication occurs, all this must be serialised and transferred from the preauth to the postauth slave
- Unfortunately, zlib has no way to serialise its state
  - But: it does provide memory allocation hooks
- OpenSSH implements a memory manager using anonymous shared memory
  - Preauth allocations shared with monitor, inherited by postauth slave
  - Monitor never uses zlib - no chance of exploit via deliberately corrupted state

Privilege separation

- Criticism: attacker may escape via kernel bugs
- Criticism: privilege separation adds complexity
  - Cleaner if designed-in, rather than retrofitted
- Criticism: OpenSSH implementation uses same buffer API as network code
  - Vulnerability in buffer code could be used to compromise both slave and monitor
  - There have been bugs in the buffer code found before
  - Alternative is to have two different RPC implementations
  - Not clear whether this would be an improvement: more heterogeneous vs. greater attack surface
- Privilege separation has reduced the criticality of all but one bugs since its introduction (early 2002)
- Second layer of checking has avoided two critical bugs

Protocol changes

- Sometimes the protocol specification requires risky things
- OpenSSH's case: activation of compression before user authentication is complete
- Result: compression code is exposed to unauthenticated users
  - attack_surface++
- Solution: change the protocol!
- Introduce zlib@openssh.com method
  - Exactly the same compression as standard zlib method
  - Only enabled **after** user has authenticated

Protocol changes

- Simple protocol change
- Simple code change (~85 lines of code, mostly mechanical)
- Backwards compatible (SSH protocol has a nice extension mechanism)
- Effectively removed ~6000 lines of code (libz) from preauth attack surface
- Criticism: OpenSSH only
- Saved us from one zlib bug since implementation (mid-2005)

Assist OS-level security measures

- Good operating systems are staring to build in attack resistance/mitigation measures
  - OpenBSD
  - Windows Vista
  - Linux (with 3rd party patches)
- Attack resistance most commonly uses *runtime randomisation*
  - Executable load address
  - Shared library load addresses
  - Stack protection cookies
  - Stackgap
  - Memory allocations

Assist OS-level security measures

- Most Unix daemons use a fork()-and-service model
  - accept() -> fork() -> do work -> exit()
  - Simple and robust
  - Unfortunately all randomisations are applied *once* - per daemon instance
- OpenSSH solution: self-reexecution
  - fork() -> **exec(sshd)** -> do work -> exit()
  - Result: each connection receives all randomisations that the OS provides
  - Additional benefit: no leakage of information from superserver to per-connection server

Assist OS-level security measures

- Some subtlety in implementation
  - Configuration must be passed from super-server to re-executed instance
- On average, re-execution doubles attack effort
  - Sampling without replacement -> sampling with replacement
- Attack becomes non-deterministic
  - No guarantee of success after N attempts
- Criticism: increases connection start-up costs
- Criticism: little benefit to platforms that do not support attack mitigation
  - It is time that they did (if Microsoft can do it, why not free operating systems?)

Future directions

- Prevent return to executable
  - If return-to-libc exploits are prevented by library randomisation, attacker can still return to the executable itself
  - E.g. to do_exec() function
  - sshd could implement additional checks to ensure that these functions cannot be called unless authentication has succeeded
  - May make some attacks more difficult

Future directions

- Separate executables for privsep
  - Current privilege separation uses single executable
  - Ease of implementation and migration, easy to disable and get pre-privsep behaviour back
  - Lots of unused code lying around in monitor
    - Return to executable attacks again
  - Separating the monitor into a dedicated executable would remove this, and make the implementation more clear
  - Some things may get harder - zlib shared memory trick may be impossible or more complicated
  - postfix[1] is a good example of a privilege separation model that uses independent cooperating processes

[1] Wietse Venema, Postfix MTA, http://www.postfix.org/

Future directions

- Pervasive testing
  - OpenSSH has a decent set of *regression* tests
  - Good for checking that your last commit didn't break anything
  - Beyond some basic sanity tests, they don't help at all with security
  - Fuzz testing is a possible approach, though a good SSH fuzzer is difficult to write
    - OUSPG has built one (no bugs found in OpenSSH :)
  - Unit tests would be better, but would be a lot of work to do retrospectively

Future directions

- Code generation
  - Lots of OpenSSH is mechanical code:
    - Packet parsing
    - Some sanity checks
    - Channel state machine
  - Idea: generate some/all of this code from a high-level description
    - High-level description will be easier to audit
    - Code generation eliminates cut-and-paste errors
  - Criticism: bugs in the code generator
  - Criticism: replacing proven and working code with untried code

Conclusion

- Relying on *never making a mistake* is doomed to failure
- Audits will not catch all mistakes
- Application developers can introduce additional security measures that reduce the likelihood and severity of bugs
- These measures are not difficult to implement and can be *retrofitted* to existing software
  - Even easier if designed in from the start

Questions?