SambaXP Keynote

Felix von Leitner, Code Blau GmbH

Who are *you*?

- IT Security consultant
- I run a small consulting company
- I get paid for ranting

So what are we doing here?

That is a very good question

In fact... let‘s start this over.

What are we doing?!

Felix von Leitner, Code Blau GmbH

Problem Statement

Our software is insecure

Countermeasures are expensive and not very effective

We need to focus our efforts!

Important Concept: TCB

TCB : Trusted Computing Base

Concept from the early 80ies.

The part of the system that is essential for security.

Idea: mv does not check permissions.
mv calls the rename() syscall, the kernel checks permissions.
The kernel is in the TCB, mv is not.

We want to be mv, not the kernel!

TCB : Trusted Computing Base

TCB is not what you ***can*** trust because it is nice and well audited.

TCB is what you ***have to*** trust because if it fails, we are all doomed.

TCB is not the good part, because we trust it.

TCB is the part that we trust, so we better make sure it is good.

Why have the concept “TCB”?

TCB: Why?

- Idea: Concentrate all our effort on the TCB!
- Make effective use of our time!

If we can get the TCB small enough, we can audit it completely!

Maybe even have a correctness proof!

Also…

- Casual glance -> piercing stare
- Objective criteria: Was this useful?
- Takes “but it feels more secure now” out of the equation

Never be fooled by “security theater” again!

Central idea of this talk:

TCB is relative to what‘s protected!

Let‘s take a web server, for example

Linux

Apache

MySQL

PHP

What does this have to do with Samba?!

I usually like to take an example that is

1. Close enough that people can recognize themselves
2. Far away enough that nobody feels personally insulted
3. Has something like PHP so we can all enjoy bashing it

Web browsers are another good example for unbound growth and security by deck chair rearranging on the Titanic.

We‘ll make it more secure!

Let‘s take a web server, for example

Linux (**Hardened**! **Full Disk Encryption**!)

Apache (runs as **user www, not root**!)

MySQL (runs as **user mysql, not root**!)

PHP (module in Apache, so also **not as root**!)

Success?

Let‘s have a look

The Linux is now hardened.

Linux is still in the TCB.

Extension 1: Data

Premise in the 80ies

Environment: multi user system with terminals.

All software running on the same system.

No separation between database and main application.

Goal: **Protect user A from user B**!

OS user accounts used to separate actual users.

Premise today

No more terminals, users come in over HTTP.

Applications running under their own UID.
Database separated from application.

Still trying to protect user A from user B!

OS user accounts used to separate parts of the TCB, not actual users!

We are doing it wrong!

80ies:

Worried about passwords.

Solution: Split /etc/passwd, put password hashes into /etc/shadow.

Make sure only root can read /etc/shadow.

Today:

Worried about passwords.

Put them into database, give PHP and Apache access to whole thing.

What are we actually trying to protect?

The data!

Consequence: **Database is in the TCB**

The database is in the TCB

In other words:

The database could be running as root.

That would not diminish overall security.

Just wait. It gets worse.

How does PHP access the database?

How does PHP access the database?

- Standard: One DB account with full access rights
- “For performance reasons”
- PHP maintains a pool of authenticated DB connections
- Every access goes over one free connection from that pool

Consequence: **PHP is in the TCB**

PHP is in the TCB

In other words:

PHP could be running as root.

That would not diminish overall security.

What about Apache?

Apache sees all the data and probably has the TLS secret keys in memory.

Apache could be running as root.

That would not diminish overall security.

So much effort, so little to show for it

Now what?

Idea: Split up PHP!

- One PHP for read-only access, one for write access
- Route SELECT statements through the read-only PHP
- Maybe have a third PHP for admin interface
- Regular “write PHP” user can only write where it must

What have we gained?

(This question needs to be asked more often!)

What have we gained?

A SQL injection in the read-only part can‘t write.

But: What did we mean by “protect the data”?

Maybe reading from the DB is a full compromise, too?
Then the read-only PHP is still in the TCB!

The write PHP is still in the TCB.

Lesson: It‘s not that easy!

Follow the ISP model!

80ies: Telekom owns network and services (BTX).

Liable for insecure services (“in the TCB”).

Now: Telekom sells network access, not liable for insecure services.

We want to our services to be where Telekom is now.

They just move data, worst case they can do is denial of service.

So let’s encrypt all the data.

Solution: Encryption!

Encrypt data in the database.

The user has the only key to their data.

Decryption with Javascript in the browser.

Result: **Web server, PHP and database not in the TCB anymore**!

But also: no server-side access to the data the user stored.

Only useful for cloud storage and file sharing scenarios.

Homomorphic Encryption!

“We’ll encrypt the data so that we can still do SELECT on them!”

My opinion:

Academic pipe dream.

Not securing anything – other than grant money.

Homomorphic Encryption!

We can structure the crypto so that some queries still work.

But: that weakens the crypto!

The more we still want to do, the weaker the crypto.

No generic solution worthwhile, crypto would be too weak.

Homomorphic Encryption!

Need to know beforehand what operations you will need to do.

Rules out data warehousing and OLAP.

For example, you could still allow “< and > on the date field”.

However: the date field is discrete numbers you could just try out.

Not much gained by “encrypting” them.

We‘ll encrypt the user name!

Typical: “XOR every letter by 0x23”

Totally worthless.

XOR with a secret key is also worthless.

Remember: We assumed the attacker hacks PHP.

The attacker can see the key.

We‘ll use public key crypto!

Also worthless as long as the attacker can guess names.

If the data contains the email address, the user name will probably be the same as the part before the @.

The public key is in my browser.

I NSA‘ed Frank‘s token.

Now I‘ll just guess user names, encrypt with the public key, and see if the token matches.

Lesson: Crypto is not the solution!

Delegate access control!

OK so we‘ll delegate access control in the other direction!

Every user gets their own DB account.

That account can see the views the user needs to see, nothing else.

The login page in PHP just passes the credentials to the DB.

Excellent idea, but…

Problem

The web server and PHP still see all user names and passwords.

With those they can access the data.

Consequence: **Apache and PHP are still in the TCB**.

Let‘s try crypto again

Need to get web server and PHP out of the TCB.

DB has key for each user. Sends out key to user.

Web server passes key from DB to user.
Browser hashes key and password and sends this back as login token.

Web server and PHP still see the token, could store it.
Apache and PHP are still in the TCB.

More crypto!

DB sends random challenge, PHP and Apache pass it on.

Problem: Apache and PHP still see the token, could do bad things with it instead of what the user asked them to do.

Apache and PHP are still in the TCB.

Even more crypto!!

Note: HTTP is stateless.

Need one challenge-response per HTTP request.

In practice we usually do login once and then have a token that grants full access. Not just to the user, also to PHP, if it was hacked and saw it.

Apache and PHP are still in the TCB.

Yet more crypto!

Thick client, does XML/JSON requests.

Each request secured with browser crypto and challenge-response.

Make a hash of the request part of the token, so it can’t be reused.

Awesome, right?

The thick client gets served by Apache, so **Apache is still in the TCB**.

One more small implementation detail

Databases generally don‘t do challenge-response.

So you‘d have to implement that in a proxy.

This kind of proxy would be implemented in PHP.

PHP is still in the TCB!

Have we gained anything?

Only if the XMLRPC/SOAP/JSON middleware is smaller than Apache+PHP.

In practice you usually see some monstrous JVM there.

It is usually even larger than Apache+PHP would have been.

Don‘t forget: The goal was to **minimize** the TCB! ☺

The whole approach is a train wreck! ☹

Extension 2: Network

Let‘s look at servers, not processes

- Network
  - Router
  - Load balancer / TLS endpoint
  - Switches
- Frontend
  - Apache
  - PHP
- Backend
  - Java middleware
  - Database

What of that is in the TCB?

Sees unencrypted data? Is in the TCB!

- Network
  - Router
  - Load balancer / TLS endpoint
  - Switches
- Frontend
  - Apache
  - PHP
- Backend
  - Java middleware
  - Database

We are approaching this wrong

What exactly are we worried about?

Attacker sees plaintext data?

No!

Attacker sees plaintext data
of other users!

… and can store / send it out!

Solution: One process per user

Prevent storing and sending

Now we are getting somewhere!

- Split off “unimportant” parts (like UI)
- Handle connections in separate, short lived processes
- Enforce separation between processes (different UIDs)
- Prevent writing to disk or outgoing connection except to the backend (firewall, read-only mount / chroot)
- One DB account per web account, with minimal rights

Result: **The software below the web server not in the TCB anymore!**

And now?

Network

- Router
- Load balancer (TLS now terminated in presentation layer)
- Switches (sees only encrypted traffic)

Frontend

- Web server, presentation layer and TLS endpoint

Backend

- Middleware
- Database

80% of the code no longer in the TCB!

Well, 80% of **our** code.

(bugs in Apache, OpenSSL, the
Java back-end or the database still kill us)

Extension 3: Sandboxing

I don’t trust this software!

Reduce the damage it can do!

- UID != 0
- chroot()
- setrlimit with RLIMIT_NOFILE, hard limit 0

Finer control possible with ACLs or things like SELinux.

“This process can open /etc/resolv.conf but nothing else from /etc”

The downside

Admin can’t configure rules for a complex application.

Usually resorts to a training mode.

Training mode fails to train error handling code paths that didn’t happen during training.

Self-Service Sandboxing!

- Process configures self-limiting rules by itself
- “I never need to call listen() again”
- “If I do, shoot me in the head”

Now we have all the tools to reduce the TCB!

If we can prevent a component from seeing or storing/sending out other people’s data, it no longer needs to be trusted.

Broker Architecture

Before:

- Web server can open files for reading.
- Web server can accept connections (for HTTP).
- Web server can initiate connections (for FastCGI, database).
- Web server can write to the file system (for logs).

All of this is needed.

Broker Architecture

Step 1: Web server writes logs to stdout, not file system.

Stdout is a pipe to another process which does the logging.
(e.g. multilog, https://cr.yp.to/daemontools.html)

Broker Architecture

Now:

- Web server can open files for reading.
- Web server can accept connections (for HTTP).
- Web server can initiate connections (for FastCGI).
- ~~Web server can write to the file system (for logs).~~

25% progress!

Broker Architecture

Fork off copy of process right after start.

Maintain a Unix Domain socket to that process.

To open a file or socket, send a message over this socket.

The other process checks the request, and if it is OK, it does the operation and sends the descriptor back over the Unix Domain socket.

Broker Architecture: Downside

IPC costs performance.

For common operations (like file system access) it may be better to use chroot/read-only mounts and not go via the broker.

Result: We can now outlaw calls to socket() and open() in the main process (and obviously ptrace, kill, etc)

What have we gained?

If we can ensure that the main process does not keep plain text data in memory, it can be removed from the TCB.

But the broker always stays in the TCB.

How do we prevent that?

Fork a new process for each request!

Broker-Architektur

After:

- ~~Web server can open files for reading.~~
- ~~Web server can accept connections (for HTTP).~~
- ~~Web server can initiate connections (for FastCGI).~~
- ~~Web server can write to the file system (for logs).~~
- Web server can ask broker for help.

Progress: >90%! (The broker stays in the TCB)

Result

Network

- Router
- Load balancer (TLS endpoint now in presentation layer)
- Switches (see only encrypted traffic)

Frontend

- Web server, presentation layer, broker and TLS endpoint

Backend

- Middleware
- Database

Conclusion

Conclusion

TCB can be minimized, but only relative to specific attack scenarios.

For SQL injection, Apache and PHP can be removed from the TCB.

- Either each HTTP user gets their own db user
- Or use validating proxy servers (surprisingly hard).

Conclusion

TCB can be minimized, but only relative to specific attack scenarios.

For Cross Site Scripting, PHP can be removed from the TCB.

- Content Security Policy
- Subresource Integrity

Conclusion

TCB can be minimized, but only relative to specific attack scenarios.

For file system accesses, networking and IPC, Apache and PHP can be taken out of the TCB.

- Chroot / jail / filesystem namespace
- SELinux / AppArmor / Systrace
- Seccomp / Capsicum / pledge

Conclusion

This would be a huge win if applied everywhere.

Audits could then focus on the other bug classes.

However, there is no way around having good, clean code, and on doing source code audits on it.

Strive to make sure audits are possible by minimizing the amount of code that has to be audited.

Howto (recap)

Don‘t design your app and then expect the OS to help you.

You‘ll find the OS does not help much.

Look at what the OS provides, and then design your app around that.

You‘ll find that you can delegate all the hard stuff to the OS.

Short lived processes: no more memory leaks.

Minimum rights: no more horrible exploits.

Thanks for your time!

Questions?

You can also e-mail me under felix-sambaxp@codeblau.de