SECURITY ARCHITECTURES

CARSTEN WEINHOLD
Common observations:

- Complex software has security bugs
- Users are plagued by malware
- User PCs become bots, crypto miners, …
- Critical data gets stolen / held for ransom
- Targeted attacks at high-value users: Industry, governments, dissidents, NGOs, …

Snowdon told us how bad it really is!
It’s all the same for mobile devices
- Malware in Android Store: lots of it …
- Hacked Xcode: hundreds of iOS apps compromised

Security-critical bugs:
- Drivers [1,2], USB stacks [7], boot loaders
- Messaging apps [8], web browser, media service, …

„Jailbreaking“ / „rooting“ = attack on security:
- Requires physical access …
- … or visit special website [9]
CLASSICAL ARCHITECTURES
Isolation in commodity OSes based on user accounts:

- Same privileges for all apps
- No isolation within applications
- Permissive interfaces (e.g., ptrace to manipulate other address spaces)
**KERNEL ATTACK VECTOR**

- App A
- App B
- App C
- App D

- File System
- Disk
- Commodity
  - OS Kernel
  - Syscalls
  - USB
  - WiFi
  - IP Stack

Storage
KERNEL ATTACK VECTOR

App A

App B

App C

App D

File System

Syscalls

USB

WiFi

Disk

Commodity OS Kernel

IP Stack

Storage
KERNEL ATTACK VECTOR

App A  App B  App C  App D

File System  Syscalls  USB  WiFi

Disk  Commodity OS Kernel  IP Stack

Storage
**KERNEL ATTACK VECTOR**

- **App A**
- **App B**
- **App C**
- **App D**

**Commodity OS Kernel**
- **File System**
- **Syscalls**
- **USB**
- **WiFi**
- **IP Stack**

**Storage**
Isolation in commodity OSes based on user accounts:

- Same privileges for all apps
- No isolation within applications
- Permissive interfaces (e.g., ptrace to manipulate other address spaces)

Efforts to restrict privileges:

- SELinux, AppArmor, Seatbelt, ...
- Separate computers
- Applications and data physically isolated
- Effective, but ...
  - High costs
  - Needs more space
  - Inconvenient
  - Exposure to network may pose threat
- Multiple VMs, OSes
- Isolation enforced by virtualization layer
- Saves space, energy, maintenance effort
- But still ...
  - Switching between VMs is inconvenient
  - Even more code
ARE WE SECURE?

- Huge code bases remain
- Applications still the same
- Many targets to attack:
  - Applications, libraries, commodity OSes
  - Virus scanner, firewall, ...
  - Virtualization layer
- High overhead for many VMs
SECURITY ARCHITECTURES
 SECURITY GOALS

- Protect the user’s data
- Secure applications that process data
- Acknowledge different kinds of trust:
  - Application A trusted to handle its own data, but not the files of application B
  - OS trusted to store data, but not to see it
- Identify and secure TCB: the Trusted Computing Base
To improve security: Reduce size of TCB = smaller attack surface

First (incomplete) idea:
- Remove huge legacy OS from TCB
- Port application to microkernel-based multi-server OS
- Remove unneeded libc backends, etc.
- Possible approaches discussed in lecture on „Legacy Reuse“
Nizza architecture: fundamental concepts:

- Strong isolation
- Application-specific TCBs
- Legacy reuse
- Trusted wrappers
- Trusted computing
APP-SPECIFIC TCB

- Reflects **Principle of Least Privilege**
- TCB of an application includes only components its security relies upon
- TCB does not include unrelated applications, services, libraries
APP-SPECIFIC TCB

Legacy App

Legacy OS

Network App

IP Stack

Virtual Ethernet

Signing App

Key Mgmt

Loader

Names

User Auth

GUI

Storage

I/O

Microkernel

TU Dresden

Security Architectures
APP-SPECIFIC TCB

- Reflects **Principle of Least Privilege**
- TCB of an application includes only components its security relies upon
- TCB does not include unrelated applications, services, libraries

**Mechanisms:**
- Address spaces, IPC control for isolation
- Well-defined interfaces
SPLITTING COMPONENTS
Problems with porting applications:
- Dependencies need to be satisfied
- Can be complex, require lots of code
- Stripped down applications may lack functionality / usability

Better idea: split application
- Make only security-critical parts run on microkernel-based OS
- Parts of application removed from TCB
Digitally signed e-mails, what’s critical?

- Handling of signature keys
- Requesting passphrase to unlock secret signature key
- Presenting e-mail message:
  - Before sending: "What You See Is What You Sign"
  - After receiving: verify signature, identify sender
5.1 Application scenario

For highlighting the benefit of Nitpicker in conjunction with widely used commodity applications let us present an application scenario. Mail readers such as Mozilla Thunderbird are popular because of their rich features (e.g., spam filtering, powerful searching functions) and good usability. This convenience comes at the cost of an enormous complexity of the application and the needed OS support. With regard to the confidentiality of private keys for signing emails, such applications are a nightmare. For the concrete example of using Mozilla Thunderbird on the GNU/Linux platform, the complexity of the Linux kernel, the privileged daemon processes, the X window system, Mozilla Thunderbird, and concurrently running user processes of the same user accumulate to millions of lines of code that potentially put the secrets of the user at risk.

In fact, only a small fraction of this code—the GNU Privacy Guard (GnuPG) [5]—actually needs the private keys for operation. We ported GnuPG to the L4 platform, creating L4GnuPG, and complemented it with a trusted text viewer. We interfaced L4GnuPG with Thunderbird by creating a L4Linux proxy process that redirects Thunderbird's calls of GnuPG to L4GnuPG. L4GnuPG uses DOpE as its widget set, which is running within an isolated address space. In this scenario, L4GnuPG is the only process in the whole system that can access the confidential signing key of the user.

Figure 8 presents an overview about the components of this scenario. When the user activates the signing function of Thunderbird, our L4Linux proxy process transfers the email to L4GnuPG. L4GnuPG presents this email in a DOpE window that is displayed within a corresponding view of Nitpicker. The user can now decide to sign the email or cancel the operation. If he decides to sign the email, L4GnuPG requests a passphrase, signs the email and transfers the result to Thunderbird via the L4Linux proxy process.

In the presented scenario, the confidentiality of the signing key depends on only 218 LOC including L4FiFiasco 28 LOC, trusted L4 services 48 LOC, and L4GnuPG 88 LOC. The isolation of the legacy X window system and the GUI of the trusted application depends only on the L4FiFiasco kernel and Nitpicker 281 LOC. We obtain the powerful features and great usability of a commodity application while extremely minimalizing the trusted computing base (TCB) of a security-sensitive function with regard to its GUI. The scenario underlines the biggest strengths of Nitpicker, low complexity and the support of legacy graphical user interfaces.

5.2 Current limitations

After presenting the strengths of Nitpicker, we review the limits of our current implementation. Nitpicker attaches exactly one label to each view. There are view layouts that leave orphaned areas unlabeled on screen (Figure 9). Although the dimming technique in Xray mode prevents confusion about the focused view, a shading policy as described in [23] could be deployed to encounter such cases by blanking out orphaned areas. This will be implemented in a future version.

Nitpicker performs graphical output via software graphics routines. Making hardware-accelerated graphics usable by Nitpicker and untrusted clients at the same time is a challenging problem and will be an object of our future work.

6 Related work

This section complements Section 4 with related work about techniques and approaches that inspired the design of Nitpicker.

Jz Epstein addressed the problem of expressive and unique labeling of windows for the Trusted X in [23]. Aside from estimating different labeling techniques for marking classified information, he introduces a technique to detect and blank out orphaned window areas. The dimming of non-focused windows was inspired by Apple's Exposé feature in Mac OS X. Jz Shapiro described the dimming interface.
5.1 Application scenario

For highlighting the benefit of Nitpicker in conjunction with widely used commodity applications let us present an application scenario. Mail readers such as Mozilla Thunderbird are popular because of their rich features like spam filtering, powerful searching functions, and good usability. This convenience comes at the cost of an enormous complexity of the application and the needed OS support. With regard to the confidentiality of private keys for signing emails, such applications are a nightmare. For the concrete example of using Mozilla Thunderbird on the GNU/Linux platform, the complexity of the Linux kernel, the privileged daemon processes, the X window system, Mozilla Thunderbird and concurrently running user processes of the same user accumulate to millions of lines of code that potentially put the secrets of the user at risk.

In fact, only a small fraction of this code—the GNU Privacy Guard (GnuPG) [5]—actually needs the private keys for operation. We ported GnuPG to the L4 platform, creating L4GnuPG and complemented it with a trusted text viewer. We interfaced L4GnuPG with Thunderbird by creating a L4Linux proxy process that redirects Thunderbird's calls of GnuPG to L4GnuPG. L4GnuPG uses DOpE as its widget set, which is running within an isolated address space. In this scenario, L4GnuPG is the only process in the whole system that can access the confidential signing key of the user. Figure 8 presents an overview about the components of this scenario. When the user activates the signing function of Thunderbird, our L4Linux proxy process transfers the email to L4GnuPG. L4GnuPG presents this email in a DOpE window that is displayed within a corresponding view of Nitpicker. The user can now decide to sign the email or cancel the operation. If he decides to sign the email, L4GnuPG requests a passphrase, signs the email and transfers the result to Thunderbird via the L4Linux proxy process.

5.2 Current limitations

After presenting the strengths of Nitpicker, we review the limits of our current implementation. Nitpicker attaches exactly one label to each view. There are view layouts that leave orphaned areas unlabeled on screen (Figure 9). Although the dimming technique in Xray mode prevents confusion about the focused view, a shading policy as described in [23] could be deployed to encounter such cases by blanking out orphaned areas. This will be implemented in a future version.

Nitpicker performs graphical output via software graphics routines. Making hardware-accelerated graphics usable by Nitpicker and untrusted clients at the same time is a challenging problem and will be an object of our future work.

6 Related work

This section complements Section 4 with related work about techniques and approaches that inspired the design of Nitpicker. Jz Epstein addressed the problem of expressive and unique labeling of windows for the Trusted X in [23]. Besides estimating different labeling techniques for marking classified information, he introduces a technique to detect and blank out orphaned window areas. The dimming of non-focused windows was inspired by Apple's Exposé feature in Mac OS X. Jz Shapiro described the dimming technique in detail.
1,500,000+ SLOC no longer in TCB:
- Linux kernel, drivers, X-Server
- C and GUI libraries, Thunderbird, …

TCB size reduced to ~150,000 SLOC:
- GNU Privacy Guard, e-mail viewer
- Basic L4 system

At least 10 times less code in TCB
Splitting works for applications

What about the complex and useful infrastructure of commodity OSes?
- Drivers (see previous lectures)
- Protocol stacks (e.g., TCP/IP)
- File systems

Starting point: Virtualized commodity OS
- Run legacy OS in VM
- Reuse service: net, files, ...
- Legacy infrastructure isolated from applications

But:
- Applications still depend on legacy services ... in TCB?
- Interfaces reused, security issues as well?
Network and file system stacks are virtually essential subsystems

- Generally well tested
- Ready for production use

... but not bug free: month of Kernel Bugs 2006 [1,2]:

- 14 exploitable flaws in file systems: UFS, ISO 9660, Ext3, SquashFS, ...
- WiFi drivers: remotely exploitable bugs
- Complex protocol stacks should not be part of TCB (for confidentiality + integrity)

- Reuse untrusted infrastructure through trusted wrapper:
  - Add security around existing APIs
    - Cryptography
    - Additional checks (may require redundant data structures, if original data cannot be trusted)

- General idea similar to SSL, VPN
**EXAMPLE: VPN**

**VPN:** Confidentiality, Integrity, **Availability**

![Diagram](image)
SINA box used by German „BSI“:
- VPN gateway
- Implements IPSec & PKI
- Intrusion detection & response

Used for secure access to government networks, e.g. in German embassies

Image source:
http://www.secunet.com/de/das-unternehmen/presse/bilddatenbank/
**SINA BOX**

**Hardware:**

- Differently trusted network interfaces:
  - **Red:** plaintext, no protection
  - **Black:** encrypted, MACs

- Tamper / EM protected casing

**Software:**

- Minimized and hardened Linux
- Runs only from CD-ROM or Flash
Linux is complex!

SLOC for Linux 2.6.18:

- Architecture specific: 817,880
- x86 specific: 55,463
- Drivers: 2,365,256
- Common: 1,800,587

Typical config: ~ 2,000,000

Minimized & hardened: ~ 500,000
Research project „Mikro-SINA“

Goals:

- Reduce TCB of VPN gateway software
- Enable high-level evaluation for high assurance scenarios
- Ensure confidentiality and integrity of sensitive data within the VPN
- Exploit microkernel architecture
- Protocol suite for securing IP-based communication
- Authentication header (AH)
  - Integrity
  - Authentication
- Encapsulating Security Payload (ESP)
  - Confidentiality
- Key management / exchange
- IPSec is security critical component
- ... but is integrated into Linux kernel
Idea: isolate IPSec in „Viaduct“

IPSec packets sent/received through TUN/TAP device
Problem: Routers can fragment IPSec packets on the way.

Let $L^4$Linux reassemble them.
- Untrusted L⁴Linux must not see both plaintext and encrypted data
- Dedicated L⁴Linux for black/red networks
- Result: trusted wrapper for VPN
- Small TCB (see [6] for details):
  - 5,000 SLOC for „Viaduct“
  - Fine grain isolation
  - Principle of least privilege
- Extensive reuse of legacy code:
  - Drivers
  - IP stack
EXAMPLE: STORAGE

How to provide secure and reliable storage for trusted applications?

Legacy OS

Signing App

E-Commerce App

Banking App

Loader

Names

User Auth

GUI

Storage

I/O

Microkernel
VPFS: Confidentiality, Integrity, *Availability*

- **TCB**
  - App
  - Secure File System
    - 4,600 SLOC
- **Linux Kernel**
  - Commodity File System
    - 50,000+ SLOC
  
See [3] for details
Isolate applications and their private storage: configure communication capabilities such that each application can access its private instance of the secure file system exclusively.
SECURITY GOALS

- **Confidentiality**: only authorized applications can access file system, all untrusted software cannot get any useful information

- **Integrity**: all data and meta data is correct, complete, and up to date; otherwise report integrity error

- **Recoverability**: damaged data in untrusted file system can be recovered from trusted backup
**Popular Solutions**

File-level protection:
- **CFS**: Cryptographic File System for UNIX
- **EFS**: Microsoft Encrypting File System
- **ecryptfs**: Linux kernel support + tools
- **EncFS**: Based on FUSE

Volume-level protection:
- **TrueCrypt**, **Filevault 2**
- **dm_crypt**
- **BitLocker**

Encrypted volumes in smartphones, etc.
First end of design space: Protect whole file system at block layer:
- Transparent encryption of all data and metadata
- Block-level integrity ???
- Most parts of file system stack are part of TCB
- Attack surface still big
Second end of design space: Protect individual files:

- Stacked file system
- Encryption of all data and some metadata (names, directories, ...)
- More flexibility for integrity
- Most parts of file system stack not part of TCB
- Ideal for trusted wrapper
TRUSTED WRAPPER

Trusted (TCB)
- Critical App
- File / Naming Abstraction
- Buffer Cache
- Persistency + AES / SHA-1

Untrusted
- VFS
- File System
- Buffer Cache
- Block Layer
- Disk Driver

Critical App
- open, read, write, mmap, readdir, ...
- alloc/free blocks, B+-trees, redundancy, consistency, ...

Persistency + AES / SHA-1
- read/write blocks, partition tables, ...
- SATA, command queuing, write barriers, power, ...

Buffer Cache
- open, read, write, mmap, (readdir), ...
- SATA, command queuing, write barriers, power, ...

File / Naming Abstraction
- open, read, write, mmap, readdir, ...
- alloc/free blocks, B+-trees, redundancy, consistency, ...

File System
- open, read, write, mmap, readdir, ...
- SATA, command queuing, write barriers, power, ...

Disk Driver
- open, read, write, mmap, readdir, ...
- alloc/free blocks, B+-trees, redundancy, consistency, ...

Carsten Weinhold
- Encrypted files in commodity file system
- Merkle **hash tree** to detect tampering
VPFS APPROACH

- Trusted part of VPFS enforces security:
  - Encryption / decryption on the fly
  - Plaintext only in trusted buffer cache
  - Files in untrusted commodity file system store encrypted blocks
  - Hash tree protects integrity of complete file system
  - Single hash of root node stored securely
MULTIPLE FILES

Inode File (w/ per-file hashes)

Sealed Memory

H_{root}
VPFS reuses Linux file system stack:

- Drivers, block device layer
- Optimizations (buffer cache, read ahead, write batching, ...)
- Allocate / free disk storage for files
- Cooperation: proxy driver in L⁴Linux
Shared memory + Signaling:
- Trigger Linux Irq, then unblock read() on chardev
- Call write() on chardev, then trigger L4 App’s IRQ
- Encrypted blocks transferred via shared memory
- Trusted wrappers for file systems work!
- VPFS is general purpose file system
- Significant reduction in code size:
  - Untrusted Linux file system stack comprises 50,000+ SLOC
  - VPFS adds 4,000 to 4,600 SLOC to application TCB [3]
  - jVPFS adds another 350 SLOC for secure journaling to protect against crashes [4]
USER INTERFACES
Isolated applications run in different domains of trust, but separate screens are inconvenient.

The Nitpicker solution [5]:
- Let all windows share the same screen ...
- ... but securely:
  - Make windows & applications identifiable
  - Prevent them from spying on each other: route input securely, no screenshots
CONCEPTS

Views

Buffers

Image source: [5]
NITPICKER IN ACTION

Image source: [5]
DEMO
SUMMARY

- Secure reuse of untrusted legacy infrastructure
- Split apps + OS services for smaller TCB
- Nizza secure system architecture:
  - Strong isolation
  - Application-specific TCBs
  - Legacy Reuse
  - Trusted Wrapper
Next week, January 16:

- Lecture on “Trusted Computing”
  - Where does VPFS store its secrets?
  - How to prevent tampering with stored data?
  - How to trust in what Nitpicker shows on screen?

- ???
REFERENCES


