Untrusted Code Security and Syscall Interception

Fall 2024

R. Sekar

Untrusted Code

- May be untrustworthy
  - Intended to be benign, but may be full of vulnerabilities
  - These vulnerabilities may be exploited by attackers (or other malicious processes) to run malicious code
- Or, may directly be malicious: may use
  - Obfuscation
    - Code obfuscation
    - Anti-analysis techniques
    - Use of vulnerabilities to hide behavior
  - (Behavioral) evasion
    - Actively subvert enforcement mechanisms

Untrusted Code

- Security is still defined in terms of policies
  - But enforcement mechanisms need to be stronger in order to defeat a strong adversary.

Reference Monitors

- Security policies can be enforced by reference monitors (RM)
  - Key requirements
    - Complete mediation
    - (If interaction with user is needed) Trusted path
- With benign code, we typically assume that it won't actively evade enforcement mechanisms
  - We can possibly maintain security even if there are ways to subvert the checks made by the RM

Types of Reference Monitors

- External RM
  - RM resides outside the address space of untrusted process
  - Relies on memory protection
    - Protect RM's data from untrusted code Limit access to RM's code
- Inline RM
  - Policy enforcement code runs within the address space of the untrusted process
  - Cannot rely on traditional hardware-based memory protection

Policies and Mechanisms for Untrusted Code

- Isolation
  - Two-way isolation
    - Chroot jails
    - Userid-based isolation, e.g., Android apps
    - Virtual machines
  - One-way isolation
    - Read access permitted, but write access denied
- System-call sandboxing
  - Linux seccomp, seccomp-bpf and eBPF
  - Delegation
- Information flow

chroot jails

- Makes the specified directory to be the root
  - Process (and its children) can no longer access files outside this directory
- Requires root privilege to chroot
  - For security, relinquish root privilege after chroot
  - All programs, libraries, configuration and data files used by this process should be within this chroot'ed directory
- Isolation limited to file system
  - e.g., it does not block interprocess interactions
  - For this reason, chroot jail is useful mainly to limit privilege escalation; but the mechanism is insecure against malicious code.
  - *Combine with other mechanisms*, e.g., seccomp.

Userid based isolation

- Create a new userid for running untrusted code
  - Real user's userid is not used, so the "Trojan horse" problem of altering permissions on user's files is avoided
- Android uses one userid for each app
  - Default permissions are set so that each app can read and write only the files it owns (except a few system directories)
- Protects against malicious interprocess interactions
  - kill, ptrace, ...
- Better than chroot, but still insufficient against malicious code
  - Can subvert benign processes by creating malicious files that may be accidentally consumed by them
    - Many sandbox escape techniques work this way

One-way isolation

- Full isolation impacts usability
  - Untrusted applications are unable to access user's files
  - Makes it difficult to use nonmalicious untrusted applications
- One-way isolation
  - Untrusted application can read any data, but writes are limited
    - Cannot overwrite user files
    - More importantly, benign applications don't ever see untrusted files –Eliminates the possibility of accidental compromise

One-way isolation

- Key issues:
  - Ensuring consistent view
    - Application creates a file and then reads it, or lists the directory
    - Inconsistencies typically lead to application failures
  - Failures due to denied write permission
    - Can overcome by creating a private copy of the file
- Both issues overcome using copy-on-write file system
- Note:
  - does not protect against loss of confidential data (without additional policies)
  - Securing user interactions is still a challenge

Securing Untrusted Code using Information Flow

- Untrusted code = low integrity, benign code = high integrity.
- Enforce the usual information flow policy that
  - Deny low integrity subject's writes to high integrity objects
    - Prevents "active subversion"
  - Deny high integrity subject's read of low integrity objects
    - Prevents "passive subversion"
    - Fooling a user (or a benign application) to perform an action, e.g., click an icon on desktop.
    - Exploit a benign process, e.g., benign image viewer compromised by reading a malicious image file.
- Can provide strong guarantee of integrity.
  - Not subject to "sandbox escapes."
- Usability issues still need to be addressed.

External Reference Monitors

- System-call based RMs
- Linux Security Modules (LSM)
- AppArmor

System-call based RMs

- OSes already implement RMs to enforce OS security policies
  - Most aspects of policy are configured (e.g., file permissions), while the RM mainly includes mechanisms to enforce these policies
- But these are typically not flexible enough or customizable
- More powerful and flexible policies may be realized using a customized RM
- System-calls provide a natural interface at which such a customized RM can reside and mediate requests.

Why Focus on System Calls

- Complete mediation: All security-related actions of processes effected via syscalls
- Application-independent semantics
  - Enables general-purpose solutions

Why Focus on System Calls

- Complete mediation: All security-related actions of processes effected via syscalls
- Application-independent semantics
  - Enables general-purpose solutions
- What sorts of problems can we solve at the syscall interface?
  - Mitigate/block vulnerability exploitations
  - Limit the scope of damage due to untrusted code and malware
  - Enforce richer security policies (beyond file permissions)
  - Transparent application extensions
  - Process replay and debugging
  - .....

Why monitor system calls?

- Expressiveness
  - Clearly defined, semantically meaningful, well-understood and well-documented interface (except for some OSes like Windows)
  - Orthogonal (each system call provides a function that is independent of other system calls — functions that rarely, if ever, overlap)
  - Can control operations for which OS access controls are ineffective, e.g., loading modules
    - A large number of security-critical operations are traditionally lumped into "administrative privilege"
- Portability: System call policies can be easily ported across similar OSes, e.g., various flavors of UNIX

Some drawbacks of system calls

- Interface is designed for functionality
  - Several syscalls may be equivalent for security purposes, but a syscall policy needs to treat them separately
- Not all relevant operations are visible
  - For instance, syscall policies cannot control name-to-file translations
- Race conditions

Race Conditions

- Pathname based policies are prone to race conditions
  - Unless the argument data is first copied into RM, checked, and then this checked copy is used by the system call.
- The window for exploiting TOCTTOU attacks can be increased by using a large sequence of symbolic links in the name
  - Increases the time taken by `open` syscall, as it may have to traverse many symlinks.
  - By combining deep directories with multiple links, you can make the lookup take a very long time
    - ... despite the relatively small limit of 40 symlink resolutions per file lookup.

Syscall Interposition at User Level

- Library interposition
  - RM resides in the same address space
  - Advantages
    - high performance
    - Potential for intercepting higher level operations
  - Drawbacks
    - RM is unprotected, so appropriate only for benign code
- Kernel-supported interposition, with RM residing in another process
  - Advantages: Secure for untrusted code
  - Drawback: High overheads due to context switches
  - Example: ptrace interface on Linux

System call interposition approaches

- Kernel interception
  - The RM resides in the kernel
  - Advantages: high performance, secure for untrusted code
  - Drawbacks (pre-eBPF):
    - difficult to program

System-call sandboxing: `ptrace`

- Primary goal of ptrace is debugging
  - Debuggers work on arbitrary programs since the OS provides this facility
- Early on in Linux evolution, ptrace was expanded to intercept syscalls
  - Made more popular by the availability of great tools like `strace`
- *Review man page on ptrace.*

System-call sandboxing: seccomp

- Seccomp is a Linux mechanism for limiting system calls that can be made by a process
  - Processes in the seccomp sandbox can make very few system calls (exit, sigreturn, read, write)
- More secure than previous mechanisms, but greatly limits actions that can be performed by a sandboxed process
  - Useful if setup properly, e.g., in Chrome, Docker, NativeClient
- Seccomp-bpf is a more recent version that permits configurable policies
  - Allowable syscalls specified in the Berkeley packet filter language
  - Policies can reference syscall name and arguments in registers

System-call sandboxing: Seccomp-BPF

- Unfortunately, most interesting policies are out-of-scope, as they reference data in process memory, e.g., file names
  - For this reason, seccomp-bpf is not much more useful than seccomp
- eBPF: more flexible
  - Better supported for observing
  - Limiting access can break things, so less support

What is eBPF?

- History: BPF (Berkeley Packet Filters) used in `tcpdump` etc. (1990s)
  - *Safe, user-programmable packet filters that run within the kernel*
  - Very limited scope
    - Limited number of instructions
    - Limited number of hooks
- eBPF: Retain safe kernel programmability, but greatly expand scope
  - A 64-bit general-purpose virtual instruction set
  - Hooks into Linux tracepoints, kprobes, etc.
  - Safety verified before loading, then JIT'ed to native code
  - For safety and stability, runtime operations mainly limited to reads (on most deployments)

eBPF Probes

- Code snippets hook to Linux kernel's well-developed tracing/hooking infrastructure
  - Tracepoints (See `/sys/kernel/debug/tracing/events`)
  - LSM hooks, kprobes (kernel function entry and exit), XDP (network stack)

eBPF Probes

- Code snippets hook to Linux kernel's well-developed tracing/hooking infrastructure
  - Tracepoints (See `/sys/kernel/debug/tracing/events`)
  - LSM hooks, kprobes (kernel function entry and exit), XDP (network stack)
- Probes written in restricted C, compiled by LLVM
  - Verified for memory safety, no global variables or loops
  - Very minimal set of helper functions (read memory, access key/value store)

eBPF Probes

- Code snippets hook to Linux kernel's well-developed tracing/hooking infrastructure
  - Tracepoints (See `/sys/kernel/debug/tracing/events`)
  - LSM hooks, kprobes (kernel function entry and exit), XDP (network stack)
- Probes written in restricted C, compiled by LLVM
  - Verified for memory safety, no global variables or loops
  - Very minimal set of helper functions (read memory, access key/value store)
- Mainly for observability, e.g., so most hooks are limited to be read-only
  - But some can respond: LSM (block), XDP (packet rewrite)

eBPF Probes

- Code snippets hook to Linux kernel's well-developed tracing/hooking infrastructure
  - Tracepoints (See `/sys/kernel/debug/tracing/events`)
  - LSM hooks, kprobes (kernel function entry and exit), XDP (network stack)
- Probes written in restricted C, compiled by LLVM
  - Verified for memory safety, no global variables or loops
  - Very minimal set of helper functions (read memory, access key/value store)
- Mainly for observability, e.g., so most hooks are limited to be read-only
  - But some can respond: LSM (block), XDP (packet rewrite)
- Communicate with user-level via
  - Perf buffers (older mechanism, one per core), Ring buffer (newer)
  - Maps (hash tables and LRU hash tables)

eBPF Experience

- The verifier can be a pain to work with
  - Fragile system; very fussy; unintuitive error messages.
  - But overall development way faster than working directly in the kernel.
- Code works “as is” across many distributions and kernels
  - This is taken for granted at the user level.
  - But in the kernel, portability often equals manual porting.
- Occasionally, you can run into obscure errors.
  - You can typically work around, or ignore because the case is so rare.

eBPF Summary

- Code snippets hook to Linux kernel's well-developed tracing/hooking infrastructure
  - Tracepoints (See `/sys/kernel/debug/tracing/events`)
  - LSM hooks
  - kprobes (kernel function entry and exit)
  - XDP (network stack)
- Most common uses
  - performance debugging
  - network monitoring and extensions
  - security monitoring

System-call delegation

- Used in conjunction with strict syscall sandboxing
  - Key idea: Delegate dangerous system calls to a helper process
  - Helper process is trusted
    - it cannot be manipulated by untrusted process
    - can implement arbitrary, application-specific access control logic
    - avoids race conditions
- Works only if
  - System call semantics permits delegation
    - e.g., not applicable for fork or `execve` — fork is usually harmless, can use `fexecve` instead of `execve`
  - Results can be transferred back transparently to untrusted process
    - e.g., file descriptors can be sent over UNIX domain sockets using sendmsg