Secure Software Development

Countermeasures: Privilege Minimization

Daniel Gruss, Vedad Hadzic, Andreas Kogler, Martin Schwarzl, Marcel Nageler

05.12.2020

Winter 2021/22, www.iaik.tugraz.at

Table of contents

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ✓

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ✓
- Exploitation ✓

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention ☑

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ✓
- Exploitation ✓
- Privilege elevation ✓

Defender's perspective

- Vulnerability prevention ✓
- Exploit prevention ✓

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery ☑
- Exploitation ☑
- Privilege elevation ☑

Defender's perspective

- Vulnerability prevention ☑
- Exploit prevention ☑
- Privilege minimization (today)

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery
- Exploitation
- Privilege elevation

Defender's perspective

- Vulnerability prevention
- Exploit prevention
- Privilege minimization

Recap: Defense In Depth

Attacker's perspective

- Vulnerability discovery
  - buffer/integer overflow, use-after-free, format strings, type confusion
- Exploitation
- Privilege elevation

Defender's perspective

- Vulnerability prevention
- Exploit prevention
- Privilege minimization

Recap: Defense In Depth

Attacker's perspective

Vulnerability discovery

- buffer/integer overflow, use-after-free, format strings, type confusion

Exploitation

Privilege elevation

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

Privilege minimization

Recap: Defense In Depth

Attacker's perspective

Vulnerability discovery

- buffer/integer overflow, use-after-free, format strings, type confusion

Exploitation

- Data corruption, shellcode, code reuse, ROP, return-to-libc

Privilege elevation

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

Privilege minimization

Recap: Defense In Depth

Attacker's perspective

Vulnerability discovery

- buffer/integer overflow, use-after-free, format strings, type confusion

Exploitation

- Data corruption, shellcode, code reuse, ROP, return-to-libc

Privilege elevation

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

- Compiler/runtime defenses, hardware defenses

Privilege minimization

Recap: Defense In Depth

Attacker's perspective

Vulnerability discovery

- buffer/integer overflow, use-after-free, format strings, type confusion

Exploitation

- Data corruption, shellcode, code reuse, ROP, return-to-libc

Privilege elevation

- admin flag, spawn a shell, cat flag.txt, gain persistence

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

- Compiler/runtime defenses, hardware defenses

Privilege minimization

Recap: Defense In Depth

Attacker's perspective

Vulnerability discovery

- buffer/integer overflow, use-after-free, format strings, type confusion

Exploitation

- Data corruption, shellcode, code reuse, ROP, return-to-libc

Privilege elevation

- admin flag, spawn a shell, cat flag.txt, gain persistence

Defender's perspective

Vulnerability prevention

- Code quality, memory safety, type safety, error handling ...

Exploit prevention

- Compiler/runtime defenses, hardware defenses

Privilege minimization

- System call filtering, sandboxing, virtualization

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability

Recap: What if Vulnerability Prevention Fails?

Attacker triggered a vulnerability

- Part 1: Can we prevent exploitation?

Recap: What if Vulnerability Prevention Fails?

Attacker triggered a vulnerability

- Part 1: Can we prevent exploitation? → Exploit Prevention

Recap: What if Vulnerability Prevention Fails?

Attacker triggered a vulnerability

- Part 1: Can we prevent exploitation? → Exploit Prevention

Attacker gained arbitrary code execution

Recap: What if Vulnerability Prevention Fails?

Attacker triggered a vulnerability

- Part 1: Can we prevent exploitation? → Exploit Prevention

Attacker gained arbitrary code execution

- Part 2: Can we prevent further damage?

Recap: What if Vulnerability Prevention Fails?

Attacker triggered a vulnerability

- Part 1: Can we prevent exploitation? → Exploit Prevention

Attacker gained arbitrary code execution

- Part 2: Can we prevent further damage? → Privilege Minimization

Recap: What if Vulnerability Prevention Fails?

- Attacker triggered a vulnerability
  - Part 1: Can we prevent exploitation? → Exploit Prevention
- Attacker gained arbitrary code execution
  - Part 2: Can we prevent further damage? → Privilege Minimization
  - Our enemy: arbitrary code execution

Privilege Minimization

Arbitrary Code Execution I

❓ What is *arbitrary code execution*?

Arbitrary Code Execution I

❓ What is *arbitrary code execution*?

- Let’s try to define it from an attacker’s perspective

Arbitrary Code Execution I

❓ What is *arbitrary code execution*?

- Let's try to define it from an attacker's perspective

➡ Attacker can choose which code to execute

Arbitrary Code Execution I

What is *arbitrary code execution*?

- Let's try to define it from an attacker's perspective

➜ Attacker can choose which code to execute

➜ Attacker obtains feedback (results, output, side-channels ...)

Arbitrary Code Execution I

- What is *arbitrary code execution*?
  - Let's try to define it from an attacker's perspective
  - Attacker can choose which code to execute
  - Attacker obtains feedback (results, output, side-channels ...)
    - Stronger but not strictly necessary

Arbitrary Code Execution I

What is *arbitrary code execution*?

- Let's try to define it from an attacker's perspective
- Attacker can choose which code to execute
- Attacker obtains feedback (results, output, side-channels ...)
  - Stronger but not strictly necessary
- Attacker can adapt the code based on the feedback and repeat

Arbitrary Code Execution I

What is *arbitrary code execution*?

- Let's try to define it from an attacker's perspective
- Attacker can choose which code to execute
- Attacker obtains feedback (results, output, side-channels ...)
  - Stronger but not strictly necessary
- Attacker can adapt the code based on the feedback and repeat
  - Even stronger but not strictly necessary

Arbitrary Code Execution I

- What is *arbitrary code execution*?
  - Let's try to define it from an attacker's perspective
  - Attacker can choose which code to execute
  - Attacker obtains feedback (results, output, side-channels ...)
    - Stronger but not strictly necessary
  - Attacker can adapt the code based on the feedback and repeat
    - Even stronger but not strictly necessary
- What does *arbitrary* mean?

Arbitrary Code Execution I

- What is *arbitrary code execution*?
  - Let's try to define it from an attacker's perspective
  - Attacker can choose which code to execute
  - Attacker obtains feedback (results, output, side-channels ...)
    - Stronger but not strictly necessary
  - Attacker can adapt the code based on the feedback and repeat
    - Even stronger but not strictly necessary
- What does *arbitrary* mean?
  - Any statement from a given language

Arbitrary Code Execution I

- What is *arbitrary code execution*?
  - Let's try to define it from an attacker's perspective
  - Attacker can choose which code to execute
  - Attacker obtains feedback (results, output, side-channels ...)
    - Stronger but not strictly necessary
  - Attacker can adapt the code based on the feedback and repeat
    - Even stronger but not strictly necessary
- What does *arbitrary* mean?
  - Any statement from a given language
  - Native x86, Bytecode, JavaScript, WebAssembly ...

Arbitrary Code Execution II

- Is arbitrary code execution always bad?

Arbitrary Code Execution II

? Is arbitrary code execution always bad?

- Well, it depends on the use case

Arbitrary Code Execution II

? Is arbitrary code execution always bad?

- Well, it depends on the use case

☹ Attacker: run arbitrary malicious payloads

Arbitrary Code Execution II

❓ Is arbitrary code execution always bad?

- Well, it depends on the use case
- ☹ Attacker: run arbitrary malicious payloads
- ☺ Browser: run responsive Websites → arbitrary JavaScript

Arbitrary Code Execution II

- ? Is arbitrary code execution always bad?
  - Well, it depends on the use case
  - ☹ Attacker: run arbitrary malicious payloads
  - ☺ Browser: run responsive Websites $\rightarrow$ arbitrary JavaScript
  - ☺ Browser: run WebApps $\rightarrow$ arbitrary WebAssembly

Arbitrary Code Execution II

❓ Is arbitrary code execution always bad?

- Well, it depends on the use case

☹ Attacker: run arbitrary malicious payloads

☺ Browser: run responsive Websites → arbitrary JavaScript

☺ Browser: run WebApps → arbitrary WebAssembly

☺ Android: run Apps → arbitrary Dalvik Bytecode

Arbitrary Code Execution II

❓ Is arbitrary code execution always bad?

- Well, it depends on the use case

☹ Attacker: run arbitrary malicious payloads

☺ Browser: run responsive Websites → arbitrary JavaScript

☺ Browser: run WebApps → arbitrary WebAssembly

☺ Android: run Apps → arbitrary Dalvik Bytecode

☺ Cloud computing: run a kernel → arbitrary native x86 instructions

Arbitrary Code Execution III

- If properly confined, arbitrary code execution is secure

Arbitrary Code Execution III

☞ If properly confined, arbitrary code execution is secure

- Browser: Website/Webapp shall not be able to compromise other tabs or the browser process

Arbitrary Code Execution III

If properly confined, arbitrary code execution is secure

- Browser: Website/Webapp shall not be able to compromise other tabs or the browser process
- Android: App shall not be able to compromise other Apps or the kernel

Arbitrary Code Execution III

☞ If properly confined, arbitrary code execution is secure

- Browser: Website/Webapp shall not be able to compromise other tabs or the browser process
- Android: App shall not be able to compromise other Apps or the kernel
- Cloud computing: Customer shall not be able to attack other customers or the cloud hypervisor

Arbitrary Code Execution III

- If properly confined, arbitrary code execution is secure
  - Browser: Website/Webapp shall not be able to compromise other tabs or the browser process
  - Android: App shall not be able to compromise other Apps or the kernel
  - Cloud computing: Customer shall not be able to attack other customers or the cloud hypervisor
- Goal: **privilege minimization**

Arbitrary Code Execution III

☞ If properly confined, arbitrary code execution is secure

- Browser: Website/Webapp shall not be able to compromise other tabs or the browser process
- Android: App shall not be able to compromise other Apps or the kernel
- Cloud computing: Customer shall not be able to attack other customers or the cloud hypervisor

☞ Goal: **privilege minimization**

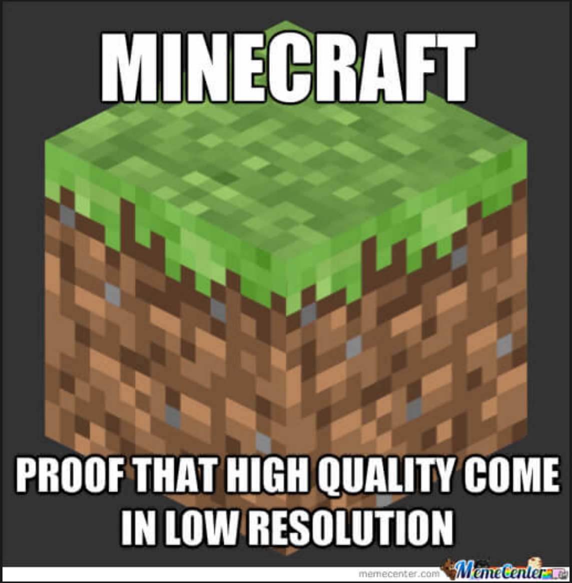
- Same privilege isolation (e.g., App vs. App)

Arbitrary Code Execution III

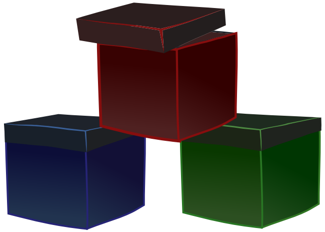
- If properly confined, arbitrary code execution is secure
  - Browser: Website/Webapp shall not be able to compromise other tabs or the browser process
  - Android: App shall not be able to compromise other Apps or the kernel
  - Cloud computing: Customer shall not be able to attack other customers or the cloud hypervisor
- Goal: **privilege minimization**
  - Same privilege isolation (e.g., App vs. App)
  - Cross-privilege isolation (e.g., App vs. kernel)

Think inside boxes ...

Think inside boxes ...



Privilege Minimization Overview



Everything is a box

- Make boxes as small as possible (Compartmentalization)
- A box shall have minimal permission (Isolation)
- "Principle of least privileges"

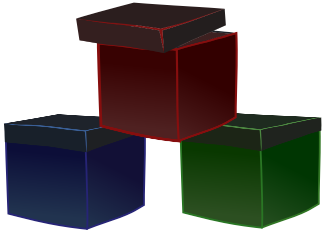
Compartmentalization

- Break large boxes into smaller boxes
- Virtual machines, processes, libraries, functions ...
- Mostly manual effort

Isolation

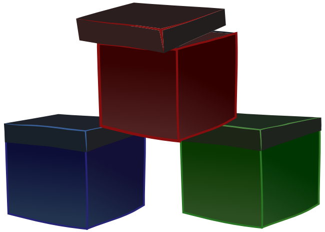
- Isolate boxes from each other
- Safeguard all interfaces
  - File permissions, network firewall ... system calls

Privilege Minimization Techniques



Isolation techniques

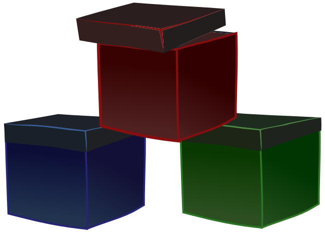
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing

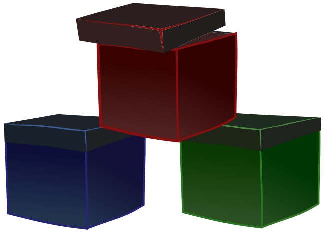
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
- Process Sandboxing

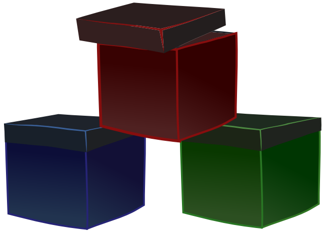
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
- Process Sandboxing
- Virtualization

Privilege Minimization Techniques



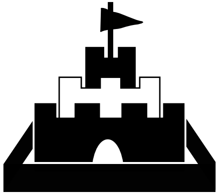
Isolation techniques

- In-process Sandboxing
- Process Sandboxing
- Virtualization
- Enclaves

In-process Sandboxing

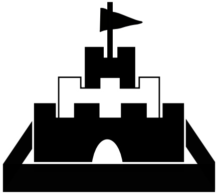
ME AFTER ESCAPING THE SANDBOX
imgflip.com

In-process Sandboxing



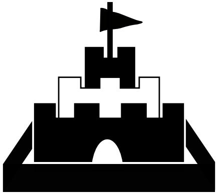
- Goal: confine parts of an application

In-process Sandboxing



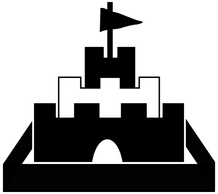
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...

In-process Sandboxing



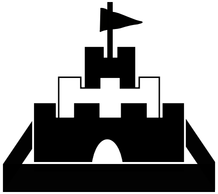
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Idea: Software-generated sandbox via

In-process Sandboxing



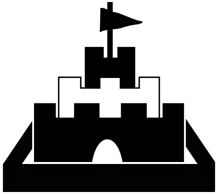
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Idea: Software-generated sandbox via
  - Interpretation

In-process Sandboxing



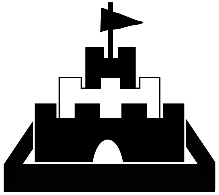
- Goal: confine parts of an application
    - E.g., dangerous plugins, libraries, user-provided code ...
- Idea: Software-generated sandbox via
    - Interpretation
    - Compilation

In-process Sandboxing



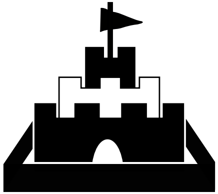
- Goal: confine parts of an application
  - E.g., dangerous plugins, libraries, user-provided code ...
- Idea: Software-generated sandbox via
  - Interpretation
  - Compilation
  - (Binary rewriting)

In-process Sandboxing: Interpretation I



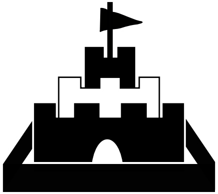
- Dangerous code is not executed natively but interpreted

In-process Sandboxing: Interpretation I



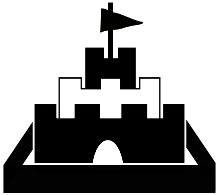
- Dangerous code is not executed natively but interpreted
  - E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...

In-process Sandboxing: Interpretation I



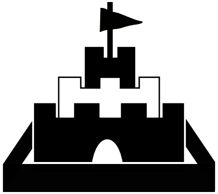
- Dangerous code is not executed natively but interpreted
  - E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...
- Interpreter

In-process Sandboxing: Interpretation I



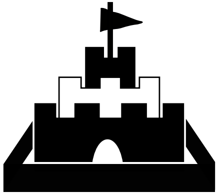
- Dangerous code is not executed natively but interpreted
  - E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...
- Interpreter
  - executes code of "virtual machine" by evaluating e.g., bytecode

In-process Sandboxing: Interpretation I



- Dangerous code is not executed natively but interpreted
  - E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...
- Interpreter
  - executes code of "virtual machine" by evaluating e.g., bytecode
  - provides hooks (callbacks) to do e.g., system calls

In-process Sandboxing: Interpretation I



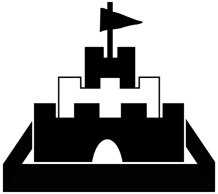
Dangerous code is not executed natively but interpreted

- E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...

Interpreter

- executes code of "virtual machine" by evaluating e.g., bytecode
- provides hooks (callbacks) to do e.g., system calls
- restricts functionality by refusing access to sensitive resources

In-process Sandboxing: Interpretation I



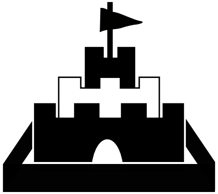
Dangerous code is not executed natively but interpreted

- E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...

Interpreter

- executes code of "virtual machine" by evaluating e.g., bytecode
- provides hooks (callbacks) to do e.g., system calls
- restricts functionality by refusing access to sensitive resources
  - Memory of the interpreter or the runtime

In-process Sandboxing: Interpretation I



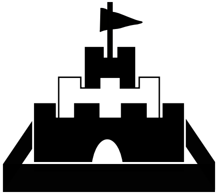
Dangerous code is not executed natively but interpreted

- E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...

Interpreter

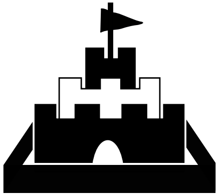
- executes code of "virtual machine" by evaluating e.g., bytecode
- provides hooks (callbacks) to do e.g., system calls
- restricts functionality by refusing access to sensitive resources
    - Memory of the interpreter or the runtime
    - System calls

In-process Sandboxing: Interpretation I



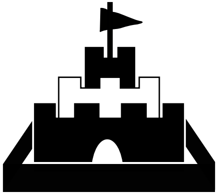
- Dangerous code is not executed natively but interpreted
  - E.g., Java, JavaScript, WebAssembly, Lua, Python, Berkeley Packet Filter (BPF) ...
- Interpreter
  - executes code of "virtual machine" by evaluating e.g., bytecode
  - provides hooks (callbacks) to do e.g., system calls
  - restricts functionality by refusing access to sensitive resources
    - Memory of the interpreter or the runtime
    - System calls
  - is an implicit sandbox

In-process Sandboxing: Interpretation II



- Properties

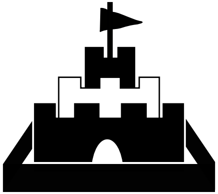
In-process Sandboxing: Interpretation II



Properties

- Interpreter can enforce very powerful and flexible policies

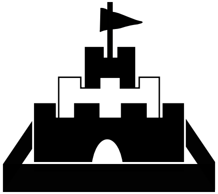
In-process Sandboxing: Interpretation II



☆ Properties

- Interpreter can enforce very powerful and flexible policies
- Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)

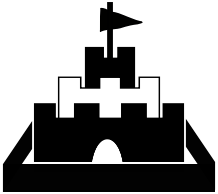
In-process Sandboxing: Interpretation II



☆ Properties

- Interpreter can enforce very powerful and flexible policies
- Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
- Slow

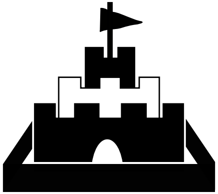
In-process Sandboxing: Interpretation II



☆ Properties

- Interpreter can enforce very powerful and flexible policies
- Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
- Slow
- Vulnerability in interpreter is fatal

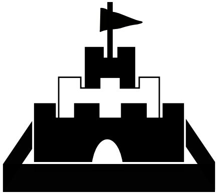
In-process Sandboxing: Interpretation II



Properties

- Interpreter can enforce very powerful and flexible policies
- Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
- Slow
- Vulnerability in interpreter is fatal
    - Prefer simple, restricted languages

In-process Sandboxing: Interpretation II

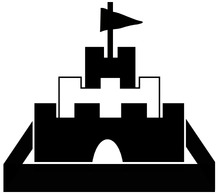


☆ Properties

- Interpreter can enforce very powerful and flexible policies
- Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
- Slow
- Vulnerability in interpreter is fatal
  - Prefer simple, restricted languages

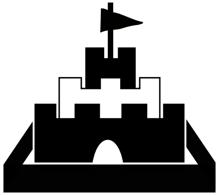
- Example: Berkeley Packet Filters (BPF)

In-process Sandboxing: Interpretation II



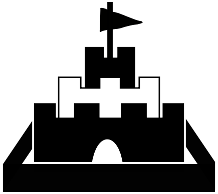
- Properties
    - Interpreter can enforce very powerful and flexible policies
    - Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
    - Slow
    - Vulnerability in interpreter is fatal
        - Prefer simple, restricted languages
- Example: Berkeley Packet Filters (BPF)
    - Interpreter runs in the kernel!

In-process Sandboxing: Interpretation II



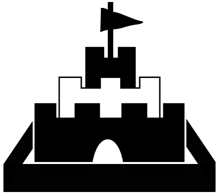
- Properties
  - Interpreter can enforce very powerful and flexible policies
  - Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
  - Slow
  - Vulnerability in interpreter is fatal
    - Prefer simple, restricted languages
- Example: Berkeley Packet Filters (BPF)
  - Interpreter runs in the kernel!
  - Use cases: network packet filtering, system call filtering

In-process Sandboxing: Interpretation II



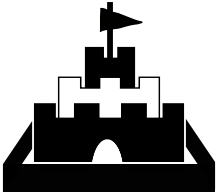
- Properties
  - Interpreter can enforce very powerful and flexible policies
  - Interpreted language typically abstracts away dangerous behavior (e.g., pointer dereference)
  - Slow
  - Vulnerability in interpreter is fatal
    - Prefer simple, restricted languages
- Example: Berkeley Packet Filters (BPF)
  - Interpreter runs in the kernel!
  - Use cases: network packet filtering, system call filtering
  - Very restricted instructions; not even turing complete

In-process Sandboxing: Compilation I



- Dangerous code is compiled to **confined native code**

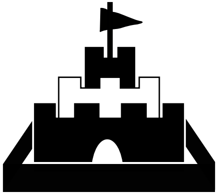
In-process Sandboxing: Compilation I



Dangerous code is compiled to **confined native code**

- Control-flow confinement, similar to CFI: Attacker cannot jump outside the sandbox

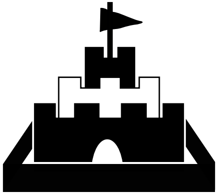
In-process Sandboxing: Compilation I



Dangerous code is compiled to **confined native code**

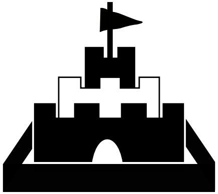
- Control-flow confinement, similar to CFI: Attacker cannot jump outside the sandbox
- Data confinement: Attacker cannot access memory outside the sandbox

In-process Sandboxing: Compilation I



- Dangerous code is compiled to **confined native code**
    - Control-flow confinement, similar to CFI: Attacker cannot jump outside the sandbox
    - Data confinement: Attacker cannot access memory outside the sandbox
- Variant 1: Compiler **introduces checks** to confine execution

In-process Sandboxing: Compilation I



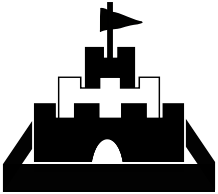
Dangerous code is compiled to **confined native code**

- Control-flow confinement, similar to CFI: Attacker cannot jump outside the sandbox
- Data confinement: Attacker cannot access memory outside the sandbox

Variant 1: Compiler **introduces checks** to confine execution

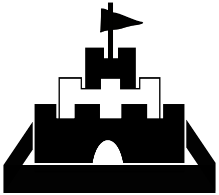
- Example: JavaScript Just-in-Time compiler

In-process Sandboxing: Compilation II



- Variant 2: Compiler **masks** all **memory accesses**

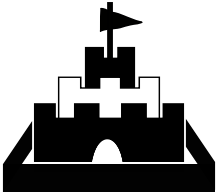
In-process Sandboxing: Compilation II



Variant 2: Compiler **masks** all **memory accesses**

- Simple logical AND operation clears upper pointer bits

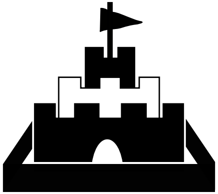
In-process Sandboxing: Compilation II



Variant 2: Compiler **masks** all **memory accesses**

- Simple logical `AND` operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory

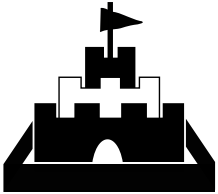
In-process Sandboxing: Compilation II



Variant 2: Compiler **masks** all **memory accesses**

- Simple logical `AND` operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory
- Also called *"Software Fault Isolation"* (SFI)

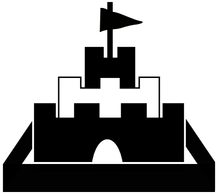
In-process Sandboxing: Compilation II



Variant 2: Compiler **masks** all **memory accesses**

- Simple logical `AND` operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory
- Also called ”*Software Fault Isolation*” (SFI)
- Example: Google Native Client (NaCl)

In-process Sandboxing: Compilation II

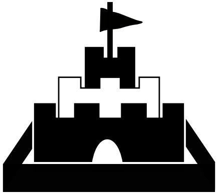


Variant 2: Compiler **masks** all **memory accesses**

- Simple logical `AND` operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory
- Also called "*Software Fault Isolation*" (SFI)
- Example: Google Native Client (NaCl)

Properties

In-process Sandboxing: Compilation II



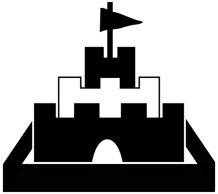
Variant 2: Compiler **masks** all **memory accesses**

- Simple logical AND operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory
- Also called "*Software Fault Isolation*" (SFI)
- Example: Google Native Client (NaCl)

Properties

- Much faster than interpretation

In-process Sandboxing: Compilation II



Variant 2: Compiler **masks** all **memory accesses**

- Simple logical `AND` operation clears upper pointer bits
- Sandbox can never access upper part of virtual memory
- Also called "*Software Fault Isolation*" (SFI)
- Example: Google Native Client (NaCl)

Properties

- Much faster than interpretation
- Requires DEP (W⊕X) to prevent bypassing the checks

Process Sandboxing

SECCOMP

Seccomp

- Observation: Most programs do not need most system calls

Seccomp

Observation: Most programs do not need most system calls

- E.g., fork, exec, prctl ...

Seccomp

- Observation: Most programs do not need most system calls
  - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls

Seccomp

- Observation: Most programs do not need most system calls
    - E.g., fork, exec, prctl ...
- Idea: block unnecessary system calls
- Implementation

Seccomp

Observation: Most programs do not need most system calls

- E.g., fork, exec, prctl ...

Idea: block unnecessary system calls

Implementation

- Program installs seccomp filters on startup

Seccomp

Observation: Most programs do not need most system calls

- E.g., fork, exec, prctl ...

Idea: block unnecessary system calls

Implementation

- Program installs seccomp filters on startup
- Seccomp supports small *Berkeley Packet Filter (BPF) programs*

Seccomp

Observation: Most programs do not need most system calls

- E.g., fork, exec, prctl ...

Idea: block unnecessary system calls

Implementation

- Program installs seccomp filters on startup
- Seccomp supports small *Berkeley Packet Filter (BPF) programs*
- Kernel does the filtering (e.g., executes the BPF program) on every system call

Seccomp

Observation: Most programs do not need most system calls

- E.g., fork, exec, prctl ...

Idea: block unnecessary system calls

Implementation

- Program installs seccomp filters on startup
- Seccomp supports small *Berkeley Packet Filter (BPF) programs*
- Kernel does the filtering (e.g., executes the BPF program) on every system call
- On a filter violation: deny syscall, send signal, kill program ...

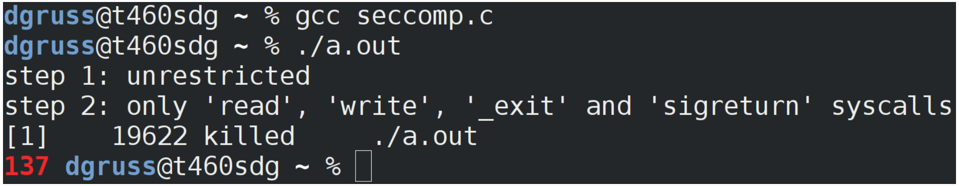
Seccomp Example I

```
#include <stdio.h>         /* printf */
#include <sys/prctl.h>     /* prctl */
#include <linux/seccomp.h> /* seccomp's constants */
#include <unistd.h>        /* dup2: just for test */

int main() {
  printf("step 1: unrestricted\n");
  prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT); // Enable filtering
  printf("step 2: only 'read', 'write', '_exit' and 'sigreturn' syscalls\n");
  dup2(1, 2); // redirect stderr to stdout
  printf("step 3: !! YOU SHOULD NOT SEE ME !!\n");
  return 0;
}
```

https://blog.yadutaf.fr/2014/05/29/introduction-to-seccomp-bpf-linux-syscall-filter/

Seccomp Example II

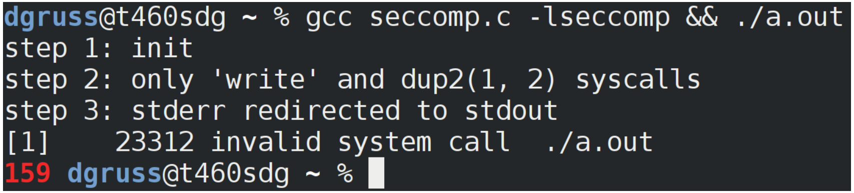


```
dgruss@t460sdg ~ % gcc seccomp.c
dgruss@t460sdg ~ % ./a.out
step 1: unrestricted
step 2: only 'read', 'write', '_exit' and 'sigreturn' syscalls
[1]    19622 killed     ./a.out
137 dgruss@t460sdg ~ % 
```

Seccomp Example III

```c
int main() {
  printf("step 1: init\n");
  prctl(PR_SET_NO_NEW_PRIVS, 1);
  prctl(PR_SET_DUMPABLE, 0);      // ptrace on this process / childs is not allowed
  scmp_filter_ctx ctx;
  ctx = seccomp_init(SCMP_ACT_KILL);                          // blacklist everything
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(rt_sigreturn), 0);    // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);            // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);      // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);            // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);           // whitelist
  seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(dup2), 2,             // whitelist
                    SCMP_A0(SCMP_CMP_EQ, 1), SCMP_A1(SCMP_CMP_EQ, 2)); // whitelist
  seccomp_load(ctx);
  printf("step 2: only 'write' and dup2(1, 2) syscalls\n");
  dup2(1, 2);     // redirect stderr to stdout
  printf("step 3: stderr redirected to stdout\n");
  dup2(2, 42);    // redirect stderr to stdout
}
```

Seccomp Example IV



```
dgruss@t460sdg ~ % gcc seccomp.c -lseccomp && ./a.out
step 1: init
step 2: only 'write' and dup2(1, 2) syscalls
step 3: stderr redirected to stdout
[1]    23312 invalid system call  ./a.out
159 dgruss@t460sdg ~ %
```

Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`

Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`

Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

→ Very convenient to use :)

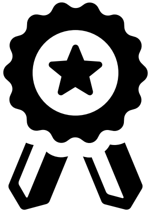
Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

→ Very convenient to use :)

- Upload your wrapper binary at `https://challenges.sasectf.student.iaik.tugraz.at/secwrap/index.php`



Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

$\rightarrow$ Very convenient to use :)

- Upload your wrapper binary at `https://challenges.sasectf.student.iaik.tugraz.at/secwrap/index.php`
- If it is correct, you will get the flag

Challenge #9 - Secwrap

Write a secure wrapper binary

- Usage: `./secwrap <command>`
- The wrapper shall start the program specified by `<command>`
- Anything `<command>` does may not be allowed to create new processes!

→ Very convenient to use :)

- Upload your wrapper binary at `https://challenges.sasectf.student.iaik.tugraz.at/secwrap/index.php`
- If it is correct, you will get the flag
- Test system is Ubuntu 20.04.1 LTS, kernel 4.19.0-11

Sandboxing with Seccomp

- Sandbox process runs dangerous code

Sandboxing with Seccomp

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC

Sandboxing with Seccomp

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC
  - Minimal filter: only allow required IPC system calls

Sandboxing with Seccomp

- Sandbox process runs dangerous code
- Monitor process interacts with sandbox via IPC
    - Minimal filter: only allow required IPC system calls
- Example: Google *sandbox2*

  `https://developers.google.com/sandboxed-api/`

Seccomp Summary

☆ Properties

Seccomp Summary

★ Properties

- Protect system call interface

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened
  - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened
  - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments
  - Enhanced filtering is impossible

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened
  - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments
  - Enhanced filtering is impossible
  - E.g., checking for strings, sanitizing paths, dereferencing pointers

Seccomp Summary

Properties

- Protect system call interface
- Filters can only be specialized but not tightened
    - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments
    - Enhanced filtering is impossible
    - E.g., checking for strings, sanitizing paths, dereferencing pointers

How do we know which system calls are needed by libc functions such as `pthread_create` ? Implementation defined!

Seccomp Summary

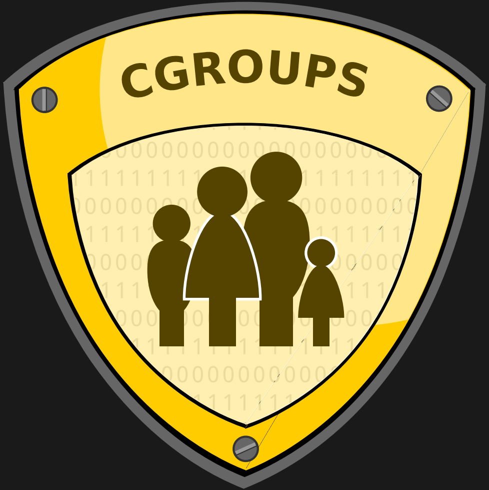
Properties

- Protect system call interface
- Filters can only be specialized but not tightened
  - Attacker cannot manipulate/unload existing filters
- Filter: simple arithmetic operations on system call arguments
  - Enhanced filtering is impossible
  - E.g., checking for strings, sanitizing paths, dereferencing pointers

How do we know which system calls are needed by libc functions such as `pthread_create` ? Implementation defined!

How can we virtualize resources?

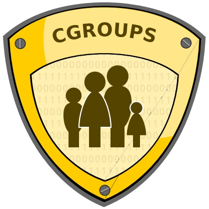
Virtualization


CGROUPS

Control groups (cgroups)

- Idea: Manage resource usage of a group of processes (and all its children)

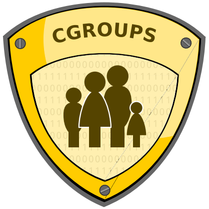
Control groups (cgroups)



Idea: Manage resource usage of a group of processes (and all its children)

- Memory, CPU time, networking, disk I/O ...

Control groups (cgroups)



Idea: Manage resource usage of a group of processes (and all its children)

- Memory, CPU time, networking, disk I/O ...
- Set limits / priorities

Control groups (cgroups)

- Idea: Manage resource usage of a group of processes (and all its children)
  - Memory, CPU time, networking, disk I/O ...
  - Set limits / priorities
- Properties

Control groups (cgroups)

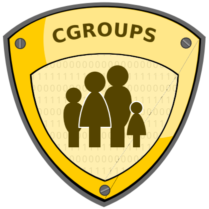
Idea: Manage resource usage of a group of processes (and all its children)

- Memory, CPU time, networking, disk I/O ...
- Set limits / priorities

Properties

- Can prevent some Denial-of-Service (DoS) attacks

Control groups (cgroups)



Idea: Manage resource usage of a group of processes (and all its children)

- Memory, CPU time, networking, disk I/O …
- Set limits / priorities

Properties

- Can prevent some Denial-of-Service (DoS) attacks
- Cannot prevent privilege escalation

NAMESPACES

Namespaces

Idea: Namespace hides (virtualizes) resources from processes

Namespaces

💡 Idea: Namespace hides (virtualizes) resources from processes

- Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`

Namespaces

Idea: Namespace hides (virtualizes) resources from processes

- Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
- How? Namespace translates resource identifiers

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:
  - Inside namespace: `uid=0(root)`, `path=/f.txt`

Namespaces

- Idea: Namespace hides (virtualizes) resources from processes
  - Various namespaces: `mnt`, `pid`, `net`, `ipc`, `uts` (hostname), `user`
  - How? Namespace translates resource identifiers
- Examples:
  - Inside namespace: `uid=0(root), path=/f.txt`
  - Outside namepsace: `uid=1000(ssd), path=/home/ssd/f.txt`

Docker Containers

Idea: combine 'em all: Docker containers

Docker Containers

Idea: combine 'em all: Docker containers

- See also Linux Containers (LXC)

Docker Containers

- Idea: combine 'em all: Docker containers
    - See also Linux Containers (LXC)
- Docker automagically

Docker Containers

- 💡 Idea: combine 'em all: Docker containers
    - See also Linux Containers (LXC)
- Docker automagically
    - creates namespaces and cgroups

Docker Containers

Idea: combine 'em all: Docker containers

- See also Linux Containers (LXC)

- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp

Docker Containers

- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties

Docker Containers

- Idea: combine 'em all: Docker containers
    - See also Linux Containers (LXC)
- Docker automagically
    - creates namespaces and cgroups
    - configures seccomp
- Properties
    - Virtualization of software resources (files, processes, users ...)

Docker Containers

- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Virtualization of software resources (files, processes, users ...)
  - Enforced via kernel

Docker Containers

- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Virtualization of software resources (files, processes, users ...)
  - Enforced via kernel
  - Fast

Docker Containers

- Idea: combine 'em all: Docker containers
    - See also Linux Containers (LXC)
- Docker automagically
    - creates namespaces and cgroups
    - configures seccomp
- Properties
    - Virtualization of software resources (files, processes, users ...)
    - Enforced via kernel
    - Fast
    - Limited to compatible kernels

Docker Containers

- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Virtualization of software resources (files, processes, users ...)
  - Enforced via kernel
  - Fast
  - Limited to compatible kernels
  - Security depends on proper configuration

Docker Containers

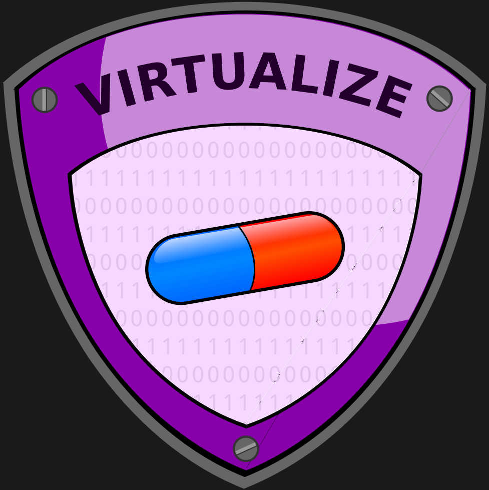
- Idea: combine 'em all: Docker containers
    - See also Linux Containers (LXC)
- Docker automagically
    - creates namespaces and cgroups
    - configures seccomp
- Properties
    - Virtualization of software resources (files, processes, users ...)
    - Enforced via kernel
    - Fast
    - Limited to compatible kernels
    - Security depends on proper configuration
        - E.g., privileged vs. unprivileged containers

Docker Containers

- Idea: combine 'em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Virtualization of software resources (files, processes, users ...)
  - Enforced via kernel
  - Fast
  - Limited to compatible kernels
  - Security depends on proper configuration
    - E.g., privileged vs. unprivileged containers
  - Kernel is shared between containers and host

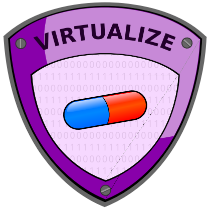
Docker Containers

- Idea: combine ’em all: Docker containers
  - See also Linux Containers (LXC)
- Docker automagically
  - creates namespaces and cgroups
  - configures seccomp
- Properties
  - Virtualization of software resources (files, processes, users ...)
  - Enforced via kernel
  - Fast
  - Limited to compatible kernels
  - Security depends on proper configuration
    - E.g., privileged vs. unprivileged containers
  - Kernel is shared between containers and host
  - What if one container compromises the host kernel?

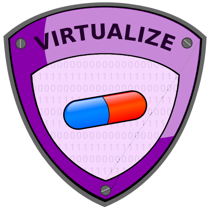

VIRTUALIZE

Full System Virtualization

- Idea: fake the entire system

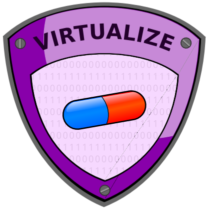


Full System Virtualization



- Idea: fake the entire system
  - Virtualization of hardware resources (memory, CPU, peripherals ...)

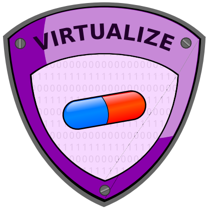
Full System Virtualization



💡 Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Full System Virtualization

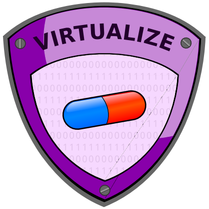


Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

Full System Virtualization



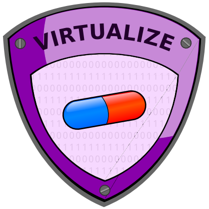
Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor

Full System Virtualization



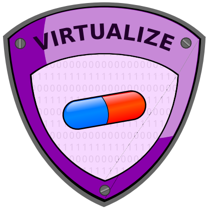
Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor
  - Xen, VMware, VirtualBox, Hyper-V, Qemu ...

Full System Virtualization



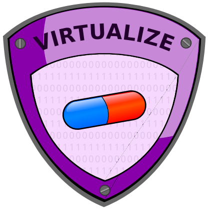
Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor
  - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
- Typically hardware-accelerated

Full System Virtualization



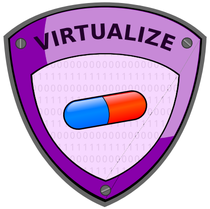
Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor
  - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
- Typically hardware-accelerated
- See other courses ...

Full System Virtualization



Idea: fake the entire system

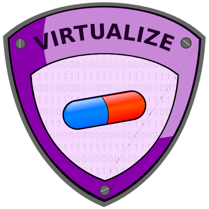
- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor
  - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
- Typically hardware-accelerated
- See other courses ...

What if VM compromises hypervisor?

Full System Virtualization



Idea: fake the entire system

- Virtualization of hardware resources (memory, CPU, peripherals ...)
- System runs many isolated virtual machines

Implementation

- Managed by hypervisor
  - Xen, VMware, VirtualBox, Hyper-V, Qemu ...
- Typically hardware-accelerated
- See other courses ...

What if VM compromises hypervisor?

Is there an end to this recursive problem?

ENCLAVES

Enclaves

- 👁 Observation: Sandboxes follow hierarchical ring model

Enclaves

Observation: Sandboxes follow hierarchical ring model

- Higher rings (kernel space) have strictly higher privileges

Enclaves

👁 Observation: Sandboxes follow hierarchical ring model

- Higher rings (kernel space) have strictly higher privileges
- Lower rings (user space) need to fully trust higher rings

Enclaves

👁 Observation: Sandboxes follow hierarchical ring model

- Higher rings (kernel space) have strictly higher privileges
- Lower rings (user space) need to fully trust higher rings
- Vulnerability in higher ring is fatal

Enclaves

Observation: Sandboxes follow hierarchical ring model

- Higher rings (kernel space) have strictly higher privileges
- Lower rings (user space) need to fully trust higher rings
- Vulnerability in higher ring is fatal

Idea: build a reverse sandbox: Enclaves

Enclaves

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)

Enclaves

Observation: Sandboxes follow hierarchical ring model

- Higher rings (kernel space) have strictly higher privileges
- Lower rings (user space) need to fully trust higher rings
- Vulnerability in higher ring is fatal

Idea: build a reverse sandbox: Enclaves

- Only trust enclave code (and hardware)
- Distrust all non-enclave code

Enclaves

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)
  - Distrust all non-enclave code
    - Host application, kernel, hypervisor

Enclaves

- Observation: Sandboxes follow hierarchical ring model
  - Higher rings (kernel space) have strictly higher privileges
  - Lower rings (user space) need to fully trust higher rings
  - Vulnerability in higher ring is fatal
- Idea: build a reverse sandbox: Enclaves
  - Only trust enclave code (and hardware)
  - Distrust all non-enclave code
    - Host application, kernel, hypervisor
  - Example: Intel Software Guard Extensions (SGX)

Enclaves

★ Properties

Enclaves

☆ Properties

- Enclaves protect a piece of secure code / data

Enclaves

★ Properties

- Enclaves protect a piece of secure code / data
- Enclaves cannot sandbox untrusted code

Enclaves

Properties

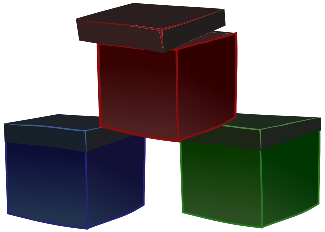
- Enclaves protect a piece of secure code / data
- Enclaves cannot sandbox untrusted code
- Can be (mis)used for Digital Rights Management (DRM), hiding malware

Enclaves

- Properties
  - Enclaves protect a piece of secure code / data
  - Enclaves cannot sandbox untrusted code
  - Can be (mis)used for Digital Rights Management (DRM), hiding malware
- Are we (too) secure now?

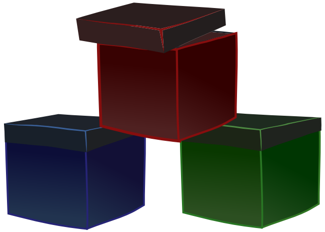
Summary & Outlook

Privilege Minimization Overview



💡 Everything is a box

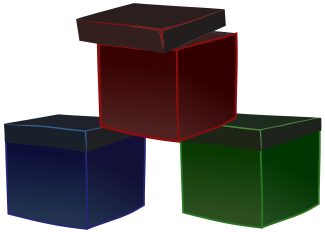
Privilege Minimization Overview



💡 Everything is a box

- **Compartmentalization**: Make boxes as small as possible

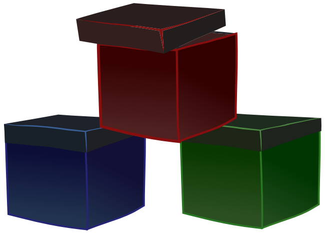
Privilege Minimization Overview



💡 Everything is a box

- **Compartmentalization**: Make boxes as small as possible
- **Isolation**: A box shall have minimal permission

Privilege Minimization Overview

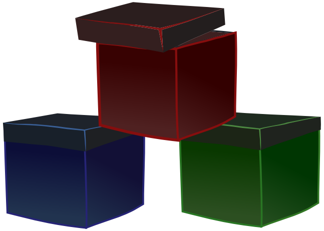


Everything is a box

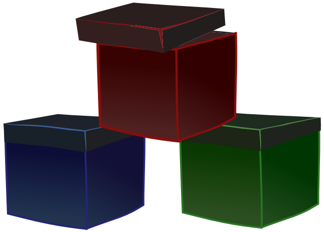
- **Compartmentalization**: Make boxes as small as possible
- **Isolation**: A box shall have minimal permission
- "Principle of least privileges"

Privilege Minimization Techniques

Isolation techniques



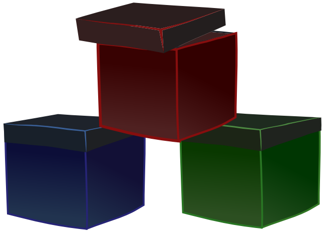
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing

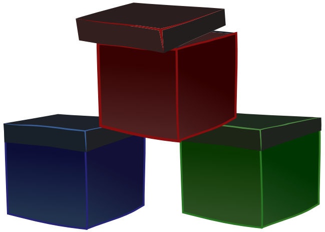
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation

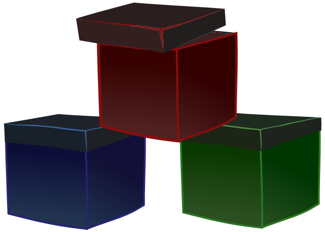
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation

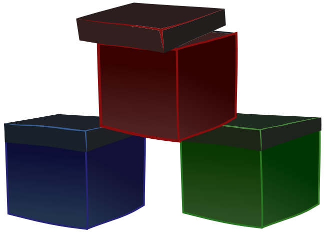
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
    - Interpretation
    - Compilation
- Process Sandboxing

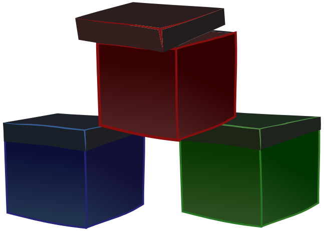
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation
- Process Sandboxing
  - Seccomp

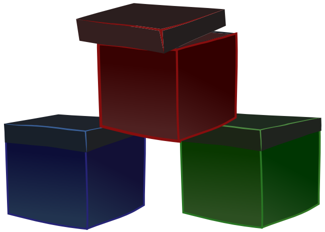
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation
- Process Sandboxing
  - Seccomp
- Virtualization

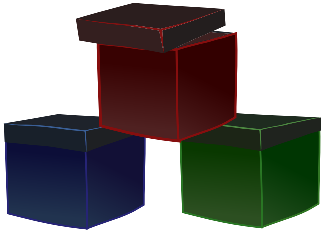
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation
- Process Sandboxing
  - Seccomp
- Virtualization
  - Docker container = seccomp + control groups + namespaces

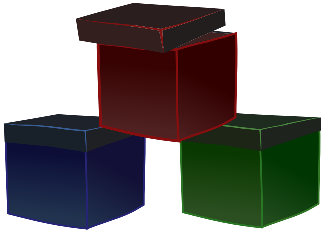
Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation
- Process Sandboxing
  - Seccomp
- Virtualization
  - Docker container = seccomp + control groups + namespaces
  - Full system virtualization

Privilege Minimization Techniques



Isolation techniques

- In-process Sandboxing
  - Interpretation
  - Compilation
- Process Sandboxing
  - Seccomp
- Virtualization
  - Docker container = seccomp + control groups + namespaces
  - Full system virtualization
- Enclaves

Save The Date: SSD Christmas Special

- Friday, 18 December, afternoon

Save The Date: SSD Christmas Special

- Friday, 18 December, afternoon
- Online show

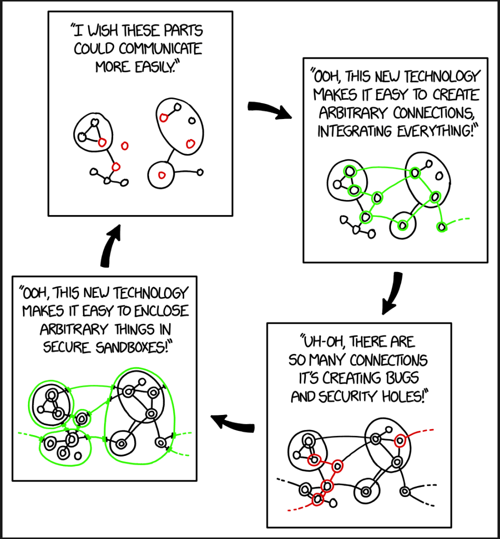
Save The Date: SSD Christmas Special

- Friday, 18 December, afternoon
- Online show
- Betting that ...

Save The Date: SSD Christmas Special

- Friday, 18 December, afternoon
- Online show
- Betting that ... you'd get surprised!

Questions?


"I WISH THESE PARTS COULD COMMUNICATE MORE EASILY."
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO CREATE ARBITRARY CONNECTIONS, INTEGRATING EVERYTHING!"
"UH-OH, THERE ARE SO MANY CONNECTIONS IT'S CREATING BUGS AND SECURITY HOLES!"
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO ENCLOSE ARBITRARY THINGS IN SECURE SANDBOXES!"