The Apple Sandbox

Dionysus Blazakis
dion@securityevaluators.com

January 11, 2011*

1 Introduction

Despite the never ending proclamations of the end of memory corruption vulnerabilities, modern software continues to fall to exploits taking advantage of these bugs. Current operating systems incorporate a battery of exploit mitigations [4][10][3] making life significantly more complex for attackers turning these bugs into attacks. Additionally, developers are becoming increasingly aware of the security implications of previously idiomatic code. Leading software publishers are teaching defensive coding techniques and have adopted an offensive mindset for product testing [9][1][8]. And yet, a single vulnerability can still provide the attacker the leverage needed to gain entry. Security researchers have disclosed multiple ways to render the mitigations ineffective [12][13][11][1] – imagine what techniques are not public. Often times, one bug can still "ruin your day" [14].

Given this problem, the inability to find *all* security relevant bugs in a system, what can be done to increase the effort required by the attacker? Lately, the most popular answer to this question has been the deployment of access control systems (sometimes called sandboxes.) Well known applications making use of this technology include Google's Chrome browser [7], Microsoft's Office 2010 Protected View [5], Apple's iOS AppStore sandboxing [6], and Adobe's upcoming Reader X [2]. Each of these applications make use of operating system specific access control systems. For Linux, a well known example is SELinux, although, other systems are available. For FreeBSD and XNU, the TrustedBSD system is used. On Windows, the access control enforcement is on the kernel object level with inherited permissions — there is no monolithic system for access control like the other operating systems.

The goal of these systems is to mitigate post-code-execution exploitation by breaking the application into tightly restricted pieces (possibly processes.) For example, the HTML

*The latest version of this paper is always available at `http://www.semantiscope.com/research/BHDC2011`.

[1]Yeah, I cited myself.

parser or Javascript engine in a web browser does not need to spawn new processes or read "/etc/password." Ideally, the developer should have the option of restricting the legal operations of a process. These valid operations are recorded in a policy as specified by the access control system. The interface for restricting a process and the format for this policy specification differs among the various access control systems. The differences in these mechanisms impacts the ease of use and flexibility of the final sandbox.

In this paper, we describe the design, implementation and usage of the Apple XNU Sandbox framework. The Sandbox framework, previously codenamed "Seatbelt", provides fine-grained access control via Scheme policy definitions. The Sandbox is implemented as a policy module for the TrustedBSD mandatory access control (MAC) framework. The Sandbox framework adds significant value by providing a user-space configurable, per-process policy on top of the TrustedBSD system call hooking and policy management engine.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the entire system. Section 3 describes the public interface and the utility function provided by the OS. Next, Section 4 walks through the details of the userspace libraries used to turn policies into sandbox syscall arguments for installing a sandbox. After the userspace interface is fully explored, Section 5 begins by briefly describing the TrustedBSD interface and how the sandbox implements this interface. Next, each kernel extension is examined. Section 5.1, documents the `Sandbox.kext` extension. In this section, the sandbox system calls are documented and the binary format of the profiles is specified. Section 5.2 examines the regular expression engine kernel extension used by the sandbox. The functions used by the sandbox are documented and the regular expression binary format is specified (since it is a subformat used in the sandbox profile binary format).

2 Overview

As mentioned in the introduction, both OS X and iOS operating systems provide an access control system current known as the Apple Sandbox. The Sandbox system is made up of: a set of userspace library functions for initializing and configuring the sandbox for each process, a Mach server for handling logging from the kernel, a kernel extension using the TrustedBSD API for enforcing individual policies, and a kernel support extension providing regular expression matching for policy enforcement. Figure 1 shows the relationships between these components on OS X. In this diagram, dark boxes denote closed source components, while light boxes denote open source components.

Sandboxing an application begins with a call to the system function `sandbox_init`. This function uses the `libsandbox.dylib` library to turn a human readable policy definition (describing rules like "don't allow access to files under `/opt/sekret`") into a binary format for the kernel. This binary format is passed to the `mac_syscall` system call handled by the TrustedBSD subsystem. TrustedBSD will pass the sandbox initialization request

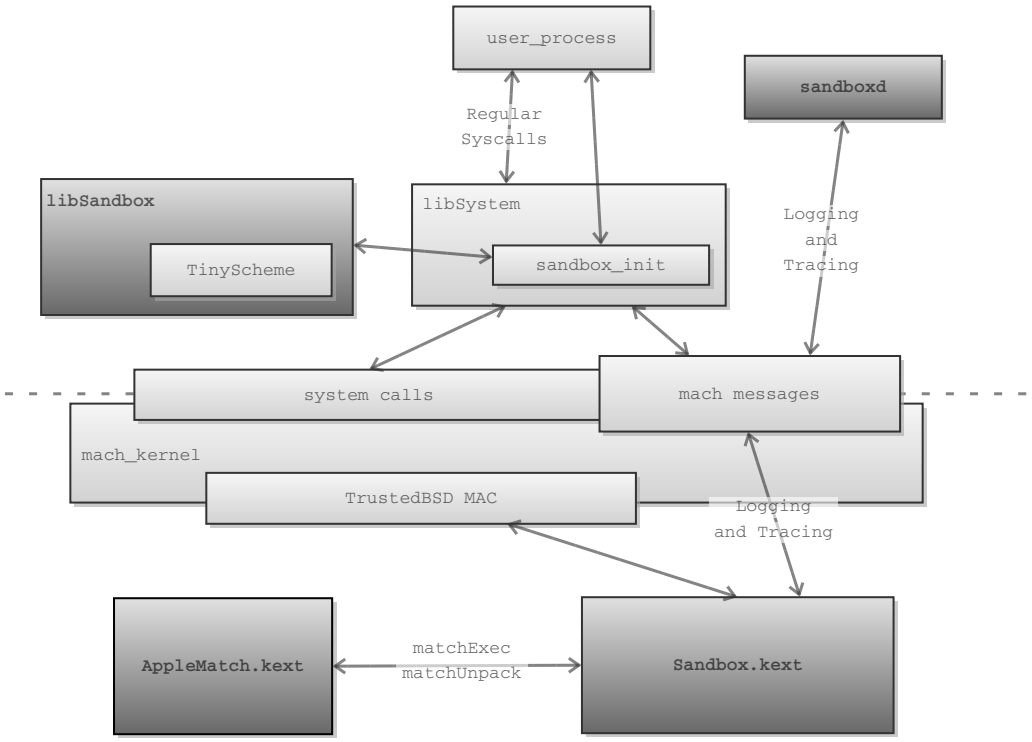

Figure 1: Apple Sandbox Overview

to the `Sandbox.kext` kernel extension for processing. The kernel extension will install the sandbox profile rules for the current process. Upon completion, a success return value will be passed back out of the kernel.

Once the sandbox is initialized, function calls hooked by the TrustedBSD layer will pass through `Sandbox.kext` for policy enforcement. Depending on the system call, the extension will consult the list of rules for the current process. Some rules (such as the example given above denying access to files under the `/opt/sekret` path) will require pattern matching support. `Sandbox.kext` imports functions from `AppleMatch.kext` to perform regular expression matching on the system call argument and the policy rule that is being checked. For example, does the file being read match the denied path `/opt/sekret/.*`? The other small part of the system is the Mach messages used to carry tracing information (such as which operations are being checked) back to userspace for logging.

The overall system is not large but has lacked any public documentation until now. Before building a system using this technology, it is important to understand how the system functions. The following sections explain in very low-level detail how the system functions.

3 Usage

The simplest way to Sandbox a process is to use the utility program `sandbox-exec`. `sandbox-exec` is wrapper that calls `sandbox_init` before a `fork` and `exec`. Command line options expose the full interface to `sandbox_init`. The interface provides three different ways to specify the access control profile: by naming a built-in profile (such as "no-internet" or "pure-computation"), by providing the path to a configuration file, or by giving the configuration directly as a string. The manpage for `sandbox-exec` is not terribly helpful (see Appendix B.1 for a copy). Within this manpage, the format of the profile configuration is not discussed nor are the pre-defined profiles named. Clearly, we must be looking in the wrong place. Let's look at a small section of the `sandbox_init` manpage (the entirety of which is in Appendix B.1):

```
AVAILABLE PROFILES
     The following are brief descriptions of each available profile.  Keep in
     mind that sandbox(7) restrictions are typically enforced at resource
     acquisition time.

     kSBXProfileNoInternet              TCP/IP networking is prohibited.

     kSBXProfileNoNetwork               All sockets-based networking is pro-
                                        hibited.

     kSBXProfileNoWrite                 File system writes are prohibited.

     kSBXProfileNoWriteExceptTemporary  File system writes are restricted to
                                        the temporary folder /var/tmp and the
                                        folder specified by the confstr(3)
                                        configuration variable _CS_DAR-
                                        WIN_USER_TEMP_DIR.

     kSBXProfilePureComputation         All operating system services are pro-
                                        hibited.
```

Ah! This one contains slightly more information — the available pre-defined profiles are listed. Let's try one of our newly found profile names:

```
fluffy:tmp dion$ sandbox-exec -n kSBXProfileNoInternet /bin/sh
sandbox-exec: profile not found
```

Rats! Unfortunately, if you know the Apple coding convention, the listed names are C string variables. To get the actual names of the built-in profiles, we'll need to write a small C program to print it out and then try `sandbox-exec` again:

```
fluffy:tmp dion$ cat <<END > /tmp/dump.c && gcc -o /tmp/dump /tmp/dump.c &&
/tmp/dump
> #include <stdio.h>
> #include <sandbox.h>
> int main(){ printf("%s\n", kSBXProfileNoInternet); return 0; }
> END
no-internet
fluffy:tmp dion$ sandbox-exec -n no-internet /bin/sh
sh-3.2$ file /tmp/dump.c
/tmp/dump.c: ASCII c program text
sh-3.2$ ping www.eff.org
PING eff.org (64.147.188.3): 56 data bytes
ping: sendto: Operation not permitted
^C
--- eff.org ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
sh-3.2$ exit
```

Success! Notice the sandbox allowed the file read of "`/tmp/dump`", but the attempted ping is denied due to the profile ("`no-internet`") we chose. The method for using any of the other four built-in profiles is clear. What about the other options to `sandbox-exec` that don't appear in the `sandbox_init` manpage? While the usage for `sandbox-exec` clearly expects a path to the profile, what is the format for this file? Unfortunately, this is where the public documentation ends; the pieces of the interface in the header, other than the configuration discussed in the manpage, are marked `__APPLE_API_PRIVATE` and left mostly undocumented. Despite the warning of an API in flux, we will soldier on and examine the rest of the `sandbox_init` API. The header at "`/usr/include/sandbox.h`" reveals that a relative path passed to `sandbox_init` will check "`/usr/share/sandbox`". Let's see if any existing profiles are available to use as an example:

```
fluffy:tmp dion$ ls /usr/share/sandbox
bsd.sb                          ntpd.sb
cvmsCompAgent.sb                portmap.sb
cvmsServer.sb                   quicklookd-job-creation.sb
fontmover.sb                    quicklookd.sb
```

```
kadmind.sb                      sshd.sb
krb5kdc.sb                      syslogd.sb
mDNSResponder.sb                xgridagentd.sb
mds.sb                          xgridagentd_task_nobody.sb
mdworker.sb                     xgridagentd_task_somebody.sb
named.sb                        xgridcontrollerd.sb
```

It appears there are plenty of examples to choose from. Let's look at `named.sb`:

```
;;
;; named - sandbox profile
;; Copyright (c) 2006-2007 Apple Inc.  All Rights reserved.
;;
;; WARNING: The sandbox rules in this file currently constitute
;; Apple System Private Interface and are subject to change at any time and
;; without notice. The contents of this file are also auto-generated and not
;; user editable; it may be overwritten at any time.
;;
(version 1)
(debug deny)

(import "bsd.sb")

(deny default)
(allow process*)
(deny signal)
(allow sysctl-read)
(allow network*)

;; Allow named-specific files
(allow file-write* file-read-data file-read-metadata
  (regex "^(/private)?/var/run/named\\.pid$"
         "^/Library/Logs/named\\.log$"))

(allow file-read-data file-read-metadata
  (regex "^(/private)?/etc/rndc\\.key$"
         "^(/private)?/etc/resolv\\.conf$"
         "^(/private)?/etc/named\\.conf$"
         "^(/private)?/var/named/"))
```

Neat, this appears to be a Scheme program. After some comments warning that we **really** should heed the advice of our elders and wait patiently for a real API specification,

the body of code is quite straightforward. Let's try writing a simple configuration:

```
fluffy:tmp dion$ sandbox-exec -p '
> (version 1)
> (allow default)
> (deny file-read-data
>     (regex #"^/private/tmp/dump\.c$"))
> ' /bin/sh
sh-3.2$ file dump
dump: Mach-O 64-bit executable x86_64
sh-3.2$ file dump.c
dump.c: cannot open: Operation not permitted
```

In this example, "`(allow default)`" sets up a blacklist style profile — unless specifically denied, a given operation is permitted by default. To test the "`file-read-data`" operation, we set up a deny filtering with literal regular expression. The transcript illustrates the filter functioning as expected, allowing the access to "`dump`" but not "`dump.c`". As the reader can probably guess, the Scheme embedded domain specific language (EDSL) Apple has constructed for profile declaration is more expressive than the simple example shown here. More details are discussed in the next section, where `libsandbox` converts the Scheme profile to a format passed to the kernel.

4 Implementation: Userspace

In this section, we will trace the path of a call to `sandbox_init` from the user process down to the syscall. Our first step is figuring out which library implements out call, `sandbox_init`. To do this, we create a simple program and use the `dyldinfo` utility:

```
fluffy:tmp dion$ cat i_call_sandbox_init.c
#include <sandbox.h>

int main(int argc, char *argv[]) {
  sandbox_init("", 0, NULL);
  return 0;
}
fluffy:tmp dion$ dyldinfo -lazy_bind i_call_sandbox_init
lazy binding information (from lazy_bind part of dyld info):
segment section          address    index  dylib            symbol
__DATA  __la_symbol_ptr  0x100001038 0x0000 libSystem        _exit
__DATA  __la_symbol_ptr  0x100001040 0x000C libSystem        _sandbox_init
```

From the output, we can see the function is implemented by `libSystem`. Given this information, the next step is to take a look in IDA. For this paper, we will always be using the 32-bit libraries. My version of `libSystem.dylib`[2] has `_sandbox_init` at `0x000330D0`. This function is mostly straightfoward, but it does give us two further directions to follow up. Under some flag values, `libsandbox` is dynamically loaded and a sequence of functions are called from this library; the sequence is made up of a "compile" function (three different "compile" functions are called from `_sandbox_init`), followed by `sandbox_apply`, and ending in `sandbox_free_profile`. For one of the flag values (2 or `SANDBOX_NAMED_BUILTIN`), the function `___sandbox_ms` is called directly and `libsandbox` is never loaded. Following this in IDA will reveal a stub for the `___mac_syscall` syscall. One thing to note, before the `libsandbox` functions are examined is the undocumented flag not listed in the `sandbox.h` header — when `flags` is set to 0, the `profile` argument is interpreted as the full profile string, presumably this is how the `-p` switch is implemented in `sandbox-exec`.

Now, we can load `libsandbox.dylib` in IDA and examine the five entries we are aware of from `sandbox_init`. My version of `libsandbox.dylib`[3] has `_sandbox_compile_string` at `0x000019CC`. I chose to start with `_sandbox_compile_string` because I expected it to be the simplest of the "compile" functions. It is; a quick look shows it to be a proxy call to the unexported function `_compile`. The other "compile" functions also end with a call to `_compile`. In fact, `_sandbox_compile_named` ends in a call to `_sandbox_compile_file` and `_sandbox_compile_file` ends in a call to `_compile`. Before digging into `_compile`, we will examine the other two functions called from `sandbox_init`.

`_sandbox_apply` is convoluted by the Mach setup code for Sandbox tracing support. By giving a trace directive in the sandbox profile, all kernel access control checks will be preceded by a Mach message. These Mach messages will be sent from the kernel to userspace where a helper process can log the check to disk. This is useful for bootstrapping a sandbox profile. There is even a utility program to clean the logging output; for details, see the `sandbox-simplify` manpage. We will touch briefly on the tracing support later in the kernel section, but I haven't focused much on this aspect in my analysis. Looking past the tracing cruft, `_sandbox_apply` is a proxy for the syscall stub we mentioned above (`___sandbox_ms`).

The last function, `_sandbox_free_profile`, is straightforward; it is made up of a few calls to `free`, releasing the memory allocated in the `_sandbox_compile_XXX` functions. With the last function out of the way, we have finished the first cut of functions coming from `_sandbox_init`. It appears all the interesting bits are in `_compile`, let's look at that now.

Before even looking at the disassembly, the IDA graph view shows a control flow graph shaped like the continent of South America (or so my co-worker would say). Any function with a flowgraph where this much branching logic occurs is interesting. This must be where

[2] MD5 (/usr/lib/libSystem.dylib) = 63c72b9767677e78105872763c888a8f

[3] MD5 (/usr/lib/libsandbox.dylib) = e6c6be5a6f3fa7bcde50126a93a2eb5d

all the magic happens. The first function call we witness is a call to `_scheme_init_new` – the Scheme profiles must be evaluated here. `_scheme_load_string` is the next interesting call. The first load is the "Initialization file for TinySCHEME 1.38". Following the initialization file, a Scheme stub defining the architecture for defining versions of the Sandbox profile language is loaded. You can extract a full listing of the Sandbox Profile Language (SBPL) Scheme files from "`strings /usr/lib/libsandbox.dylib`". The stub mentions each SBPL version will define two Scheme scripts: a prelude and a body. Immediately following the stub in the library is the `_sbpl1_scm` string – the body of SBPL version 1. Before discussing the main SBPL library code, the comment at the top of the SBPL stub describes the end result of a profile evaluation:

```
;;;;;; Sandbox Profile Language stub
;;; This stub is loaded before the sandbox profile is evaluated.  When version
;;; is called, the SBPL prelude and the appropriate SBPL version library are
;;; loaded, which together implement the profile language.  These modules build
;;; a *rules* table that maps operation codes to lists of rules of the form
;;;   RULE -> TEST | JUMP
;;;   TEST -> (filter action . modifiers)
;;;   JUMP -> (#f . operation)
;;; The result of an operation is decided by the first test with a filter that
;;; matches.  Filter can be #t, in which case the test always matches.  A jump
;;; causes evaluation to continue with the rules for another operation.  The
;;; last rule in the list must either be a test that always matches or a jump.
```

As explained in the comment, the end result is a vector(`*rules*`) of rules. On OS X 10.6.4, the first 59 entries correspond to operations (eg. `file-read-data` or `sysctl-write`). To check, logically, if an operation is permitted, the `*rules*` table is consulted. The operation code is used as an index into the table to find the rule entry for the candidate operation. For example, on OS X 10.6.4, `file-read-data` is assigned operation code 5. Suppose the 6^{th} entry of `*rules*` is `(#f . 0)`. This is a `JUMP` rule as listed in the above comment — it equates `file-read-data` with the `default` entry (which has operation code 0). The entire SBPL is a Scheme embedded domain specific language to turn policy rules into a binary decision diagram enforcing those rules. Since this decision tree is the policy that is later passed to the kernel, a Scheme interpreter is not embedded in the kernel (much to the chagrin of attackers[4]).

As mentioned above, the initialization script is from TinyScheme 1.3.8 — does `libsandbox` use TinyScheme as the interpreter or just the initialization script? Comparing with IDA or BinDiff would work, but a simple comparison of symbols is enough. The Scheme interpreter in `libsandbox` is based on TinyScheme[5]. Given this information, let's test and

[4]Or just me. I really wanted to find a Scheme interpreter in the XNU kernel.

[5]http://tinyscheme.sourceforge.net/home.html

enhance our understanding of the `*rules*` format — we will use the `display` function to display `*rules*` after evaluating a Sandbox profile.

Our first approach is using the stock TinyScheme 1.3.8 distribution. First, we build the `scheme` executable and replacing the `init.scm` provided in the distribution with the one extracted from `libsandbox`. The functions `take` and `drop` are newly defined in the `libsandbox` initialization file. Next, we will try loading the SBPL Scheme scripts on load order, adding one at a time to see what breaks. The first error we get is from loading `sbpl_1.scm`:

```
fluffy:sbpl dion$ ./scheme sbpl_stub.scm  sbpl_1_prelude.scm  sbpl_1.scm
Error: undefined sharp expression
Errors encountered reading sbpl_1.scm
```

No line numbers `;_;`. But, using gdb and the input we can track it down to the so-called sharp expression parsing. It seems Apple has added the notion of a raw string to their Scheme interpreter using a sharp expression to encode it. Just as a Python script could use `r"raw\n"`, an SBPL profile could use `#"raw\n"`. This is not built into TinyScheme 1.3.8 or 1.3.9. Reversing the differences in the parsing and adding this logic to the TinyScheme 1.3.8 `scheme.c` source results in a 21 line patch (included in the Appendix). After this change, the next error we get is:

```
fluffy:sbpl dion$ ./scheme sbpl_stub.scm sbpl_1_prelude.scm sbpl_1.scm ../sb/ntpd.sb
Error: eval: unbound variable: %version-1
Errors encountered reading ../sb/ntpd.sb
```

This is method is defined in `libsandbox.dylib`. The function loads the `sbpl_1_prelude` and `sbpl_1` scripts. For our purposes, we can just add a function returning `#f` to `sbpl_stub.scm`. Last error is:

```
fluffy:sbpl dion$ ./scheme sbpl_stub.scm sbpl_1_prelude.scm sbpl_1.scm ../sb/ntpd.sb
Error: eval: unbound variable: *params*
Errors encountered reading ../sb/ntpd.sb
```

We can define an empty parameter list in `sbpl_stub.scm`. Finally, all scripts will load. Let's add a script at the end to dump `*rules*` and run our scripts:

```
fluffy:sbpl dion$ cat dump_rules.scm
(display *rules*)
(display "\n")
fluffy:sbpl dion$ ./scheme sbpl_stub.scm sbpl_1_prelude.scm sbpl_1.scm
../sb/ntpd.sb dump_rules.scm
#( ((#t deny))
```

```
   ((#f . 0))
   ((#f . 1))
   (((filter path 0 regex ^/dev/dtracehelper$) allow) (#f . 1))
   ((#f . 1))
   (((filter path 0 regex
             ^/dev/null$
             ^(/private)?/var/run/syslog$
             ^/dev/u?random$
             ^/dev/autofs_nowait$
             ^/dev/dtracehelper$
             /\.CFUserTextEncoding$
             ^(/private)?/etc/localtime$
             ^/usr/share/nls/
             ^/usr/share/zoneinfo/
             ^/usr/lib/.*\.dylib$
             ^/usr/lib/info/.*\.so$
             ^/System/
             ^/private/var/db/dyld/
             ^(/private)?/etc/hosts\.(allow|deny)$
             ^(/private)?/var/run/ntpd\.pid$
             ^(/private)?/var/db/ntp\.drift(\.TEMP)?$
             ^(/private)?/etc/ntp\.(conf|keys)$)
     allow) (#f . 4))
...
```

(The output was manually "pretty-printed" by the author — the real output was nice and unformatted.) Hooray! It looks as described[6]. We now understand how a full Scheme profile is turned into this compact representation. What happens next? How is this sent to the kernel? One way to figure this out is to reverse `_compile`. I didn't take that route, instead I followed the blob sent via the syscall into the kernel and started taking it apart from that code. Because of this, I think we're done with userspace.

As far as I know, the only piece of the Sandbox system we haven't touched is `sandboxd`. I don't really want to get into all that. I believe it acts as a Mach server listening for the Sandbox access check tracing messages I mentioned previously, but if I'm wrong, I'll deny I ever said that. For more information, get a copy of IDA and do it yourself.

Let's go to the kernel.

[6] I became somewhat nervous after finding the sharp express error. While the patch I give seems to work, I wasn't sure what other modifications could have been done. I could have thrown it into BinDiff, but it is easier to `dlopen("libsandbox.dylib", ...)` and call the embedded TinyScheme directly. It requires some calculation to use the unexported symbols, but it's worth the piece of mind. I used this version for any other experiments I performed. Again, see the appendix for full source.

5 Implementation: Kernel

We will enter the kernel via the syscall we mentioned earlier. Let's revisit that and follow the syscall number to the kernel syscall table. The stub in `libSystem` is `___sandbox_ms`:

```
__text:00033C6C ; int __sandbox_ms(const char *policyname, int call, void *arg)
__text:00033C6C                 public ___sandbox_ms
__text:00033C6C ___sandbox_ms   proc near
__text:00033C6C                 mov     eax, 0C017Dh    ; ___mac_syscall
__text:00033C71                 call    __sysenter_trap
__text:00033C76                 jnb     short locret_33C86
__text:00033C78                 call    $+5
__text:00033C7D                 pop     edx
__text:00033C7E                 mov     edx, ds:(off_1A88E0 - 33C7Dh)[edx]
__text:00033C84                 jmp     edx
__text:00033C86 ; ---------------------------------------------------------
__text:00033C86
__text:00033C86 locret_33C86:                   ; CODE XREF: ___sandbox_ms+A j
__text:00033C86                 retn
__text:00033C86 ___sandbox_ms   endp
```

As IDA so helpfully commented, `0x000C017D` specifies syscall number 381 a.k.a. `___mac_syscall`. Here it is in syscall.master (found in `xnu-1504.7.4/bsd/kern/`[7]):

```
381     AUE_MAC_SYSCALL ALL     { int __mac_syscall(char *policy, int call,
                                  user_addr_t arg); }
```

`__mac_syscall` is implemented on line 2119 of `xnu-1504.7.4/security/mac_base.c`. The top of this source file is a copyright talking about the TrustedBSD project. Some quick work on Google explains this to be a mandatory access control (MAC) framework originally written by Robert Watson for FreeBSD. It was ported to OS X and provides the underlying syscall hooking and kernel object tagging necessary for many access control policies. A policy kernel module sets up a table of system calls and kernel structure life cycle management functions to hook and then calls the TrustedBSD API (`mac_policy_register`) to install itself. More information and multiple papers on the design of TrustedBSD can be found on the website: `http://www.trustedbsd.org/`.

`__mac_syscall` is straightforward; it looks up the policy based on a passed in string and then call then proxies the syscall to that policy module. The arguments are listed in the comment: a string for the name of the policy, an integer for the policy module to multiplex

[7] Throughout this section, I will be refering to the xnu-1504.7.4 source as downloaded from `http://www.opensource.apple.com/tarballs/xnu/xnu-1504.7.4.tar.gz`

on, and an argument pointer. The string is used to select the policy module to proxy to — we can inspect this value from userspace and use it to find the extension implementing the sandbox. Grabbing the MAC policy name used in the `sandbox_init` call is easy with gdb:

```
fluffy:tmp dion$ cat <<END > /tmp/pname.c && \
> gcc -m32 -o /tmp/policy_name_for_200_trebeck \
> /tmp/pname.c && gdb /tmp/policy_name_for_200_trebeck
> #include <sandbox.h>
> main() { sandbox_init(kSBXProfileNoInternet, SANDBOX_NAMED, NULL); }
> END
GNU gdb 6.3.50-20050815 (Apple version gdb-1472) (Wed Jul 21 10:53:12 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...
Reading symbols for shared libraries .. done

(gdb) break __mac_syscall
Breakpoint 1 at 0x8c9c6c
(gdb) run
Starting program: /private/tmp/policy_name_for_200_trebeck
Reading symbols for shared libraries +. done
Reading symbols for shared libraries ... done

Breakpoint 1, 0x907b9c6c in __sandbox_ms ()
(gdb) x/s *(int *)($esp + 4)
0x25363:  "Sandbox"
```

Oh. Ok. We also could have looked at `0x000332CB` of `libsandbox` to see the parameter. Let's look for a loaded kernel extension that uses sandbox in the name:

```
fluffy:tmp dion$ kextstat | grep -i sandbox
23    0 0x1a97000  0x8000  0x7000  com.apple.security.sandbox (0) <21 7 6 5 4 2 1>
```

Now, to find the kernel extension:

```
fluffy:tmp dion$ find /System/Library/Extensions -name '*.plist' \
> -exec grep -l 'sandbox' '{}' ';'
/System/Library/Extensions/Sandbox.kext/Contents/Info.plist
```

Finally, we have the kernel extension that implements the TrustedBSD policy for the Sandbox.

5.1 Sandbox.kext

This is the heart of the Apple Sandbox framework. As mentioned above, any kernel extension that implements a policy must register with TrustedBSD by calling `mac_policy_register`. `Sandbox.kext` does this when the kernel extension is loaded:

```
__text:0000000A _kmod_start     proc near       ; DATA XREF: __data:__realmain o
__text:0000000A           push    ebp
__text:0000000B           mov     ebp, esp
__text:0000000D           sub     esp, 18h
__text:00000010           mov     dword ptr [esp+8], offset _kmod_stop ; xd
__text:00000018           mov     dword ptr [esp+4], offset _policy_handle ; handlep
__text:00000020           mov     dword ptr [esp], offset _policy_conf ; mpc
__text:00000027           call    near ptr _mac_policy_register
__text:0000002C           add     esp, 18h
__text:0000002F           pop     ebp
__text:00000030           retn
__text:00000030 _kmod_start     endp
```

The first argument to `mac_policy_register` is a structure giving more details about the policy (including a pointer to the table of function pointers handling the syscall hooks):

```
__data:000054C0 ; struct mac_policy_conf policy_conf
__data:000054C0 _policy_conf    dd offset aSandbox      ; mpc_name
__data:000054C0                                         ; DATA XREF: _kmod_start+16 o
__data:000054C0                 dd offset aSeatbeltSandbo; mpc_fullname ; "Sandbox"
__data:000054C0                 dd offset _labelnames   ; mpc_labelnames
__data:000054C0                 dd 1                    ; mpc_labelname_count
__data:000054C0                 dd offset _policy_ops   ; mpc_ops
__data:000054C0                 dd 0                    ; mpc_loadtime_flags
__data:000054C0                 dd offset _label_slot   ; mpc_field_off
__data:000054C0                 dd 0                    ; mpc_runtime_flags
__data:000054C0                 dd 0                    ; mpc_list
__data:000054C0                 dd 0                    ; mpc_data
```

Here, we can verify the policy name is "Sandbox" as expected (given our gdb experiment above). This structure is defined on line 6136 of `xnu-1504.7.4/security/mac_policy.h`:

```
6136 struct mac_policy_conf {
```

```
6137   const char            *mpc_name;            /** policy name */
6138   const char            *mpc_fullname;        /** full name */
6139   const char            **mpc_labelnames;     /** managed label namespaces */
6140   unsigned int           mpc_labelname_count; /** number of [above] ... */
6141   struct mac_policy_ops *mpc_ops;             /** operation vector */
6142   int                    mpc_loadtime_flags;  /** load time flags */
6143   int                   *mpc_field_off;       /** label slot */
6144   int                    mpc_runtime_flags;   /** run time flags */
6145   mpc_t                  mpc_list;            /** List reference */
6146   void                  *mpc_data;            /** module data */
6147 };
```

A full explanation of the TrustedBSD MAC Framework is outside the scope of this paper. For our purposes, it suffices to know that the `mpc_ops` structure member contains a table of function pointers to hook sysem calls and kernel object (descriptors, mostly) lifecycle operations. I won't reproduce the large structure defining the operations here — it can be found on line 5787 of `xnu-1504.7.4/security/mac_policy.h`. Back to the original goal, we are trying to find the function handling the syscall. The function call used in `__mac_syscall` is named `mpo_policy_syscall` in the `policy_ops` structure. We follow the cross reference to the `policy_ops` structure to find the `mac_syscall` handler function (`_hook_policy_syscall` at `0x000003E0`):

```
__data:00004FD0         dd offset _hook_policy_syscall; mpo_policy_syscall
```

Navigating to `_hook_policy_syscall` reveals some actual processing. The first bit of the function switches on three values for the `call` argument (0 to 2). Let's write wrappers of `mac_syscall` to document the arguments and operation of the sub-syscalls:

```
#include <sys/types.h>
#include <stddef.h>

int __sandbox_ms(const char *policyname, int call, void *arg);

int sandbox_init_bytecode(void *buff, size_t len, void *unk)
{
  struct {
    user_addr_t bytecode;
    user_size_t bytecode_len;
    user_addr_t unknown;
  } args;

  args.bytecode = CAST_USER_ADDR_T(buff);
```

```
  args.bytecode_len = (user_size_t) len;
  args.unknown = CAST_USER_ADDR_T(unk);

  return __sandbox_ms("Sandbox", 0 /*call*/, &args);
}

int sandbox_init_builtin(char *policy)
{
  struct {
    user_addr_t policy;
  } args;

  args.policy = CAST_USER_ADDR_T(policy);

  return __sandbox_ms("Sandbox", 1 /*call*/, &args);
}

int sandbox_check_raw(/*out*/ int *rv, pid_t pid, char *operation,
    int filter_type, char *path)
{
  int msrv;

  struct {
    int rv;
    int success;
  } results;

  struct {
    user_addr_t result;
    pid_t pid;
    user_addr_t operation;
    user_long_t filter_type;
    user_addr_t path;
  } args;

  args.result = CAST_USER_ADDR_T(&results);
  args.pid = pid;
  args.operation = CAST_USER_ADDR_T(operation);
  args.filter_type = (user_long_t) filter_type;
  args.path = CAST_USER_ADDR_T(path);
```

```
  msrv = __sandbox_ms("Sandbox", 2 /*call*/, &args);
  if (msrv != 0)
    return msrv;

  if (rv != NULL)
    *rv = results.rv;

  return msrv;
}
```

(Yes, this is more userspace, but the details of this interface was easier to understand – for me – from the kernel side.) The first two functions correspond to the undocumented `SANDBOX_RAW` and `SANDBOX_NAMED_BUILTIN` respectively. The first function is the one we are most concerned with. This function takes a raw "compiled" profile and applies it to the current process. The second function takes a name for a built-in and applies it to the current process. Don't be confused by the two layers of named "built-in" profiles: `SANDBOX_NAMED` uses precompiled profiles stored in the userspace library (`libsandbox.dylib`), while `SANDBOX_NAMED_BUILTIN` uses precompiled profiles from the kernel extension (`Sandbox.kext`). The last sub-syscall is not for setting up a profile, but for asking if a given operation will be denied by the current profile.

Now that we undertand how to call `mac_syscall` (and exercise all paths of `_hook_policy_syscall`), we can start to reverse the meat of the function. For `call` values of 0 and 1, the same operations occur after the bytecode is copied in (in the 0 case) or the built-in bytecode is found (in the 1 case). Let's follow the action starting at `0x00000417` of `Sandbox.kext`. The first basic block copies the argument structure from userspace. The next three check that the bytecode length is sane and then malloc and fill a buffer for the bytecode in the kernel. Next, at `0x000004BB`, a structure is allocated and passed to `_re_cache_init` along with a pointer to the bytecode buffer. I'll spare you the play-by-play; this function fills a cache with unpacked regular expressions from the compiled sandbox profile. This function is the first clue to the structure of the bytecode passed to the kernel. Given the information from `_re_cache_init`, we can deduce the following structure for the profile:

```
header:
  u16_le re_offset_table_offset (in 8-byte words)
  u8 re_offset_table_count

@ re_offset_table_offset:
  u16_le re_offset[re_offset_table_count]

@ re_offset[n]
  u32_le re_size
  u8 re_bytes[re_size]
```

_re_cache_init iterates over all compiled regexes in the profile initializing (matchInit) and unpacking (matchUnpack) each. These two functions are defined in AppleMatch.kext. In the next section, we'll take a look at the format of these compiled regular expressions and the exported functions of AppleMatch.kext. For now, let's get back to the rest of the sandbox profile initialization.

Back in _hook_policy_syscall after the call to _re_cache_init completes, the next block performs a call to _sandbox_create. This function allocates a structure and stores a lock and a pointer to the structure passed to _re_cache_init (it contains the original bytecode and the regex cache). Following the initialization of this structure, _proc_apply_sandbox stores this in the policy label slot. These label slots are managed by TrustedBSD and a slot is optionally allocated when the policy is registered. This structure holds the persistent state needed for evaluating access control checks. When _proc_apply_sandbox returns, this is effectively the end of the sandbox initialization.

One function I skipped above, called twice in _proc_apply_sandbox, was _sb_evaluate. This function (finally!) does most of the interesting work for the enforcement of the profile rules. To support this claim, look at the body of the hooked syscall functions in the _policy_ops table. Most of these functions check access by calling _sb_evaluate with an operation code and a filter context. One example is _hook_mount_check_fsctl; this function calls _cred_check (which proxies a call to our function of interest – _sb_evaluate). Notice _cred_check takes an argument in edx specifying which operation to check (in this case 0x30, representing the "system-fsctl" operation when looked up in _operation_names table).

Reversing _sb_evaluate will yield a full understanding of the compiled profile format. I've summarized the file format below:

```
header:
u2 re_table_offset (8-byte words from start of sb)
u1 re_table_count (really just the low byte)
u1 padding
u2[] op_table (8-byte word offset)

ophandlers:
u1 opcode
    01: terminal
    00: non-terminal

terminal:
    u1: padding
    u1: result
      00: allow
      01: deny
```

```
non-terminal:
    u1 filter
        01: path
        02: xattr
        03: file-mode
        04: mach-global
        05: mach-local
        06: socket-local
        07: socket-remote
        08: signal
    u2 filter_arg
    u2 transition_matched
    u2 transition_unmatched
```

5.2 AppleMatch.kext

Besides walking the filter decision tree, `Sandbox.kext` doesn't do much beyond looking up data structures and checking an integer here (ports) or a bit there (file permissions). The one exception is the regular expression matching. When parsing the profile in `_hook_policy_syscall`, a major chunk of the profile is usually devoted to the regular expression cache. As we noted above, the compiled regular expressions are first passed to `matchUnpack`. Following the cross reference in `Sandbox.kext` reveals `matchUnpack` is an import. IDA doesn't know which extension was linked to provide this symbol, so which kernel extension is linked for these calls? `AppleMatch.kext` exports the symbols we're looking for:

```
fluffy:Extensions dion$ kextlibs -all-symbols Sandbox.kext 2>&1| grep matchUnpack
    _matchUnpack in /System/Library/Extensions/AppleMatch.kext (1.0.0d1)
    _matchUnpack in /System/Library/Extensions/AppleMatch.kext (1.0.0d1)
fluffy:Extensions dion$ nm -arch i386 \
> AppleMatch.kext/Contents/MacOS/AppleMatch | \
> grep matchUnpack
000006e4 T _matchUnpack
```

Looking at `AppleMatch.kext` in IDA, it is clear this is a real regular expression engine. The functions imported from this extension by `Sandbox.kext` are `matchInit`, `matchUnpack`, `matchExec`, and `matchFree`. Let's walk through each in turn.

`matchInit` creates a structure to store state between the unpacking and execution of a regular expression. The C prototype would look something like:

```
typedef void *(*m_alloc_func)(unsigned int size, const char *note);
```

```
typedef void (*m_free_func)(void *addr, const char *note);

struct matchExpr;
typedef struct matchExpr matchExpr_t;

int matchInit(matchExpr_t **m, m_alloc_func a, m_free_func f);
```

`matchInit` requires two function parameters (in addition to the address of a pointer to hold the allocated structure.) — an allocation and free function to be used by any match functions taking a `matchExpr_t`. This is convenient and adds to ease of running this code in userspace (which can be done directly – see the Appendix A for details on where to find a harness utility program).

After creating and initializing the state storage struct, a regular expression is unpacked and the internal structures needed to execute (aka regex match) the regular expression are derived. `matchUnpack` provides this functionality. The C prototype would look something like:

```
int matchUnpack(unsigned char *buffer, unsigned int length, matchExpr_t *m)
```

If you weren't paying attention, you might guess the `buffer` parameter took a nice ASCII string like `"(a|b)+(c*)"` (a regular expression!). Since everyone reading this is much more insightful and can extract meaning from the slightest of context clues, I don't need to explain that the regular expression is actually in a compiled form. While compiling the sandbox profile in `libsandbox`, the regular expressions are compiled from their ASCII representations into a binary format using a userspace version of the `AppleMatch` library named `libMatch.dylib`. We have a lot of tools to reversing this format — we have the original regex, the compiled form embedded in the sandbox profile, and the code responsible for compiling and then unpacking this format. Using all of this (and some distant memories of that theory of computation course), it becomes clear the regular expressions are converted to nondeterministic finite automata (NFA).

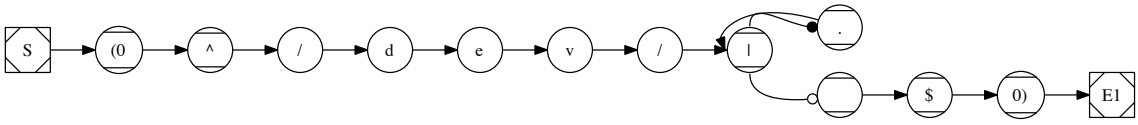

Figure 2: NFA for "`^/dev/.*$`"

See Figure 2 for an example of a regular expression turned into an NFA. This diagram was generated[8] by the `matchDiagram` function in `libMatch`. See Appendix A for details on where to find a small program that generates the GraphViz dot file for a compiled regular

[8] I only found this function **after** writing my own crappier version in Python > . <

expression. An NFA is executed from the start state and transitioning based on the input string. An NFA may split into multiple "current" states. If the end state is reached (by any of the "current" states), the NFA accepts the input. Try a few examples; it's pretty simple. This NFA traversal is done in kernel during the `matchExec` function of `AppleMatch.kext`.

The `matchExec` function uses the state structure set up by the previous `matchUnpack` function to attempt a regular expression match (really, an NFA acceptance test). The C prototype looks something like:

```
struct matchInput {
  unsigned char *start;
  unsigned char *end;
};
typedef struct matchInput matchInput_t;

int matchExec(matchExpr_t *m,
              matchInput_t *inputs,
              unsigned int *input_count,
              unsigned int *result);
```

First, this function takes a pointer to the previously initialized and loaded state structure. The next two arguments specify an array of strings to attempt matches with. The last argument is an output parameter to denote if a match was found — the return value denotes a error condition, while `result` denotes success or failure of the regular expression match.

We have discussed all relevant function calls from `AppleMatch.kext`. The next step is to describe the packed format at the byte level. In other words, how is the NFA structure encoded in the regular expression entries of the sandbox profile?

```
re:
  u4 version? (must be 1 or unpack fails)
  u4 node_count
  u4 start_node
  u4 end_node
  u4 cclass_count
  u4 submatch_count
  node nodes[]
  cclass cclasses[]

node:
  u4 type
  u4 arg
```

```
  u4 transition

cclass:
  u4 count
  u4 spans[]
```

6 Acknowledgements

I want to thank the guys at ISE for asking me all the hard questions when I was presenting early versions of this research. Andrew Case of Digital Forensics Solutions, LLC read an early version and helped polish. Dan told me my CFG looked like the continent of Africa (he was wrong). JHU Security and Privacy Group let me do a dry run and only charged me for pizza. `#formal` contains a bunch of people smarter than me that keep me up to date on real science.

A Software

All scripts mentioned in this paper are available at: `http://github.com/dionthegod/XNUSandbox`

B Manpages

B.1 sandbox-exec

```
SANDBOX-EXEC(1)           BSD General Commands Manual          SANDBOX-EXEC(1)

NAME
     sandbox-exec -- execute within a sandbox

SYNOPSIS
     sandbox-exec [-f profile-file] [-n profile-name] [-p profile-string]
                  [-D key=value ...] command [arguments ...]

DESCRIPTION
     The sandbox-exec command enters a sandbox using a profile specified by
     the -f, -n, or -p option and executes command with arguments.

     The options are as follows:

     -f profile-file
```

Read the profile from the file named profile-file.

-n profile-name
Use the pre-defined profile profile-name.

-p profile-string
Specify the profile to be used on the command line.

-D key=value
Set the profile parameter key to value.

SEE ALSO
sandbox_init(3), sandbox(7), sandboxd(8)

B.2 sandbox_init

```
SANDBOX_INIT(3)          BSD Library Functions Manual          SANDBOX_INIT(3)

NAME
     sandbox_init, sandbox_free_error -- set process sandbox

SYNOPSIS
     #include <sandbox.h>

     int
     sandbox_init(const char *profile, uint64_t flags, char **errorbuf);

     void
     sandbox_free_error(char *errorbuf);

DESCRIPTION
     sandbox_init() places the current process into a sandbox(7). The
     NUL-terminated string profile specifies the profile to be used to config-
     ure the sandbox.  The flags specified are formed by or'ing the following
     values:

     SANDBOX_NAMED           The profile argument specifies a sandbox profile
                             named by one of the constants given in the
                             AVAILABLE PROFILES section below.

     The out parameter *errorbuf will be set according to the error status.

RETURN VALUES
     Upon successful completion of sandbox_init(), a value of 0 is returned
     and *errorbuf is set to NULL.  In the event of an error, a value of -1 is
     returned and *errorbuf is set to a pointer to a NUL-terminated string
     describing the error.  This string may contain embedded newlines.  This
     error information is suitable for developers and is not intended for end
     users.  This pointer should be passed to sandbox_free_error(3) to release
     the allocated storage when it is no longer needed.

AVAILABLE PROFILES
     The following are brief descriptions of each available profile.  Keep in
     mind that sandbox(7) restrictions are typically enforced at resource
     acquisition time.
```

kSBXProfileNoInternet	TCP/IP networking is prohibited.
kSBXProfileNoNetwork	All sockets-based networking is prohibited.
kSBXProfileNoWrite	File system writes are prohibited.
kSBXProfileNoWriteExceptTemporary	File system writes are restricted to the temporary folder /var/tmp and the folder specified by the confstr(3) configuration variable _CS_DARWIN_USER_TEMP_DIR.
kSBXProfilePureComputation	All operating system services are prohibited.

SEE ALSO

sandbox-exec(1), sandbox(7), sandboxd(8)

Mac OS X July 7, 2007 Mac OS X

References

[1] Adobe Reader and Acrobat Security Initiative. http://blogs.adobe.com/asset/2009/05/adobe_reader_and_acrobat_secur.html.

[2] Adobe Reader Protected Mode. http://blogs.adobe.com/asset/2010/07/%20introducing-adobe-reader-protected-mode.html.

[3] OS X Security. http://www.apple.com/macosx/security/.

[4] PaX. http://www.grsecurity.net/.

[5] Protected View in Office 2010. http://blogs.technet.com/b/office2010/archive/2009/08/13/protected-view-in-office-2010.aspx.

[6] Security Overview: Sandboxing and the Mandatory Acccess Control Framework. http://developer.apple.com/library/ios/documentation/Security/Conceptual/Security_Overview/Concepts/Concepts.html#//apple_ref/doc/uid/TP30000976-CH203-SW1.

[7] The Chromium Projects: Sandbox Design. http://www.chromium.org/developers/design-documents/sandbox.

[8] The Cisco Secure Development Lifecycle: An Overview. http://blogs.cisco.com/security/the_cisco_secure_development_lifecycle_an_overview/.

[9] The Trustworthy Computing Security Development Lifecycle. http://msdn.microsoft.com/en-us/library/ms995349.aspx.

[10] Windows ISV Software Security Defenses. http://msdn.microsoft.com/en-us/library/bb430720.aspx.

[11] Dionysus Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. *Blackhat DC*, 2010.

[12] Tyler Durden. Bypassing PaX ASLR protection. *Phrack*, 0x0b(0x3b):0x09, 2002.

[13] Alex Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. In *Blackhat USA*, 2008.

[14] Dino Dai Zovi. One Exploit Should Not Ruin Your Day. http://trailofbits.com/2010/01/24/one-exploit-should-not-ruin-your-day/.