Project Zero

News and updates from the Project Zero team at Google

Monday, November 24, 2014

pwn4fun Spring 2014 - Safari - Part II

Posted by Ian Beer

TL;DR
An OS X GPU driver trusted a user-supplied kernel C++ object pointer and called a virtual function. The IOKit registry contained kernel pointers which were used defeat kASLR. A kernel ROP payload ran Calculator.app as root using a convenient kernel API.

Overview of part I
We finished part I with the ability to load our own native library into the Safari renderer process on OS X by exploiting an integer truncation bug in the Safari javascript engine. Here in part II we'll take a look at how sandboxing works on OS X, revise some OS X fundamentals and then exploit two kernel bugs to launch Calculator.app running as root from inside the Safari sandbox.

Safari process model
Safari's sandboxing model is based on privilege separation. It uses the WebKit2 framework to communicate between multiple separate processes which collectively form the Safari browser. Each of these processes is responsible for a different part of the browser and sandboxed to only allow access to the system resources it requires.

Specifically Safari is split into four distinct process families:

- *WebProcesses* are the renderers - they're responsible for actually drawing web pages as well as dealing with most active web content such as javascript
- *NetworkProcess* is the process which talks to the network
- *PluginProcesses* are the processes which host native plugins like Adobe Flash
- *UIProcess* is the unsandboxed parent of all the other processes and is responsible for coordinating the activity of the sandboxed processes such that a webpage is actually displayed to the user which they can interact with

The Web, Network and Plugin process families are sandboxed. In order to understand how to break out of the WebProcess that we find ourselves in we've first got to understand how this sandbox is implemented.

OS X sandboxing primitives
OS X uses the Mandatory Access Control (MAC) paradigm to implement sandboxing, specifically it uses the TrustedBSD framework. Use of the MAC sandboxing paradigm implies that whenever a sandboxed process tries to acquire access to some system resource, for example by opening a file or creating a network socket, the OS will first check: *Does this particular process have the right to do this?*

An implementation of sandboxing using TrustedBSD has two parts: firstly, hooks must be added to the kernel code wherever a sandboxing decision is required. A TrustedBSD hook looks like this:

```
/* bsd/kern/uipc_syscalls.c */
int socket(struct proc *p, struct socket_args *uap, int32_t *retval)
{
#if CONFIG_MACF_SOCKET_SUBSET
 if ((error = mac_socket_check_create(kauth_cred_get(), uap->domain,
     uap->type, uap->protocol)) != 0)
  return (error);
#endif /* MAC_SOCKET_SUBSET */
...
```

That snippet of code is from the implementation of the `socket` syscall on OS X. If MAC support has been enabled at compile time then the very first thing the `socket` syscall implementation will do is call `mac_socket_check_create`, passing the credentials of the calling processes and the domain, type and protocol of the requested socket:

```
/* security/mac_socket.h */
int mac_socket_check_create(kauth_cred_t cred, int domain, int type, int
protocol)
{
 int error;
```

Search This Blog
Search
Archives
► 2017 (15)
► 2016 (17)
► 2015 (33)
▼ 2014 (11)
► December (1)
▼ November (2)
pwn4fun Spring 2014 - Safari - Part II
Project Zero Patch Tuesday roundup, November 2014
► October (2)
► September (1)
► August (2)
► July (3)

```
 if (!mac_socket_enforce)
  return 0;

 MAC_CHECK(socket_check_create, cred, domain, type, protocol);
 return (error);
}
```

Here we see that if the enforcement of MAC on sockets hasn't been globally disabled (`mac_socket_enforce` is a variable exposed by the `sysctl` interface) then this function falls through to the `MAC_CHECK` macro:

```
/* security/mac_internal.h */
#define MAC_CHECK(check, args...) do {                \
 for (i = 0; i < mac_policy_list.staticmax; i++) {   \
  mpc = mac_policy_list.entries[i].mpc;              \
...
  if (mpc->mpc_ops->mpo_## check != NULL)            \
   error = mac_error_select(                         \
       mpc->mpc_ops->mpo_ ## check (args),           \
       error);
```

This macro is the core of TrustedBSD. `mac_policy_list.entries` (the first highlighted chunk) is a list of *policies* and the second highlighted chuck is TrustedBSD *consulting* the policy. In actual fact a policy is nothing more than a C struct (`struct policy_ops`) containing function pointers (one per hook type) and consultation of a policy simply means calling the right function pointer in that struct.

If that policy function returns `0` (or isn't implemented at all by the policy) then the MAC check succeeds. If the policy function returns a non-zero value then the MAC check fails and, in the case of this `socket` hook, the syscall will fail passing the error code back up to userspace and the rest of the `socket` syscall won't be executed.

The second part of an implementation of sandboxing using TrustedBSD is the provision of these policy modules. Although TrustedBSD allows multiple policy modules to be present at the same time in practice on OS X there's only one and it's implemented in its own kernel extension: Sandbox.kext. When it's loaded Sandbox.kext registers itself as a policy with TrustedBSD by passing a pointer to its `policy_ops` structure. TrustedBSD adds this to the `mac_policy_list.entries` array seen earlier and will then call into Sandbox.kext whenever a sandboxing decision is required.

Sandbox.kext and the OS X sandbox policy_ops

This paper from Dionysus Blazakis, this talk from Meder Kydyraliev and this reference from @osxreverser go into great detail about Sandbox.kext and its operation and usage.

Summarizing those linked resources, every process can have a unique sandbox profile. For (almost) every MAC hook type Sandbox.kext allows a sandbox profile to specify a decision tree to be used to determine whether the MAC check should pass or fail. This decision tree is expressed in a simple scheme-like DSL built from tuples of actions, operations and filters (for a more complete guide to the syntax refer to the linked docs):

```
(action operation filter)
```

- Actions determine whether a particular rule corresponds to passing or failing the MAC check. Actions are the literals `allow` and `deny`.
- Operations define which MAC hooks this rule applies to. For example the `file-read` operation allows restricting read access to files.
- Filters allow a more granular application of operations, for example a filter applied to the `file-read` operation could define a specific file which is or isn't allowed.

Here's a snippet from the WebProcess sandbox profile to illustrate that:

```
(deny default (with partial-symbolication))
...
(allow file-read*
       ;; Basic system paths
       (subpath "/Library/Dictionaries")
       (subpath "/Library/Fonts")
       (subpath "/Library/Frameworks")
       (subpath "/Library/Managed Preferences")
       (subpath "/Library/Speech/Synthesizers")
       (regex #"^/private/etc/(hosts|group|passwd)$")
...
)
```

As you can see sandbox profiles are very readable on OS X. It's usually quite clear what any particular sandbox profile allows and denies. In this example the profile is using regular expressions to define allowed file paths (there's a small regex matching engine in the kernel in AppleMatch.kext.)

Sandbox.kext also has a mechanism which allows userspace programs to ask for policy decisions. The main use of this is to restrict access to system IPC services, access to which isn't mediated by the kernel (so there's nowhere to put a MAC hook) but by the userspace daemon launchd.

Enumerating the attack surface of a sandboxed process

Broadly speaking there are two aspects to consider when enumerating the attack surface reachable from within a particular sandbox on OS X:

- Actions which are specifically allowed by the sandbox policy - these are easy to enumerate by looking at the sandbox policy files.
- Those actions which are allowed because either because the Sandbox.kext `policy_ops` doesn't implement the hook callback or because there's no hook in place at all.

The Safari WebProcess sandbox profile is located here:
`/System/Library/StagedFrameworks/Safari/WebKit.framework/Versions/A/Resources/com.apple.WebProcess.sb`

This profile uses an `import` statement to load the contents of `/System/Library/Sandbox/Profiles/system.sb` which uses the `define` statement to declare various broad sandboxing rulesets which define all the rules required to use complete OS X subsystems such as graphics or networking. Amongst others the Webprocess.sb profile uses `(system-graphics)` which is defined here in system.sb:

```
(define (system-graphics)
...
  (allow iokit-open
         (iokit-connection "IOAccelerator")
         (iokit-user-client-class "IOAccelerationUserClient")
         (iokit-user-client-class "IOSurfaceRootUserClient")
         (iokit-user-client-class "IOSurfaceSendRight"))
  )
)
```

This tells us that the WebProcess sandbox has pretty much unrestricted access to the GPU drivers. In order to understand what the `iokit-user-client-class` actually means and what this gives us access to we have to step back and take a look at the various parts of OS X involved in the operation of device drivers.

OS X kernel fundamentals

There are two great books I'd recommend to learn more about the OS X kernel: the older but still relevant "Mac OS X Internals" by Amit Singh and the more recent "Mac OS X and iOS Internals: To the Apple's Core" by Jonathan Levin.

The OS X wikipedia article contains a detailed taxonomic discussion of OS X and its place in the UNIX phylogenetic tree but for our purposes it's sufficient to divide the OS X kernel into three broad subsystems which collectively are known as XNU:

BSD

The majority of OS X syscalls are BSD syscalls. The BSD-derived code is responsible for things like file systems and networking.

Mach

Originally a research microkernel from CMU mach is responsible for many of the low-level idiosyncrasies of OS X. The mach IPC mechanism is one of the most fundamental parts of OS X but the mach kernel code is also responsible for things like virtual memory management.

Mach only has a handful of dedicated mach syscalls (mach calls them traps) and almost all of these only exist to support the mach IPC system. All further interaction with the mach kernel subsystems from userspace is via mach IPC.

IOKit

IOKit is the framework used for writing device drivers on OS X. IOKit code is written in C++ which brings with it a whole host of new bug classes and exploitation possibilities. We'll return to a more detailed discussion of IOKit later.

Mach IPC

If you want to change the permissions of a memory mapping in your process, talk to a device driver, render a system font, symbolize a crash dump, debug another process or determine the current network connectivity status then on OS X behind the scenes you're really sending and receiving mach messages. In order to find and exploit bugs in all those things it's important to understand how mach IPC works:

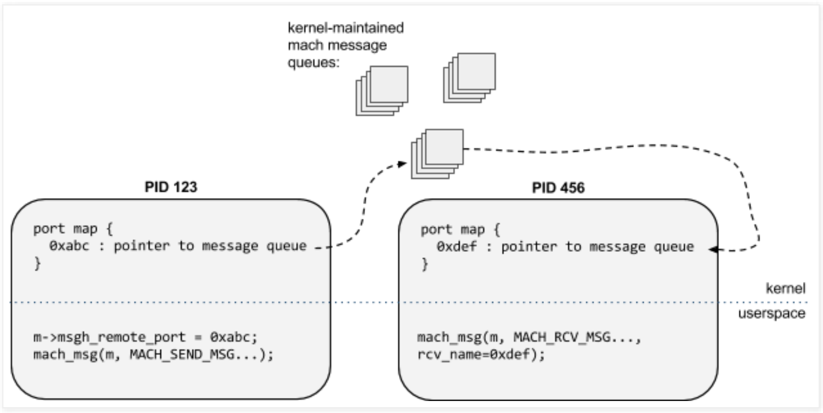
Messages, ports and queues

Mach terminology can be a little unclear at times and OS X doesn't ship with the man pages for the mach APIs (but you can view them online here.)

Fundamentally mach IPC is message-oriented protocol. The messages sent via mach IPC are known as mach messages. *Sending* a mach message really means the message gets enqueued into a kernel-maintained message queue known as a *mach port*.

Only one process can dequeue messages from a particular port. In mach terminology this process has a *receive-right* for the port. Multiple processes can enqueue messages to a port - these processes hold *send-rights* to that port.

Within a process these send and receive rights are called mach port *names*. A mach port name is used to index a per-process mapping between mach port names and message queues (akin to how a process-local UNIX file descriptor maps to an actual file):

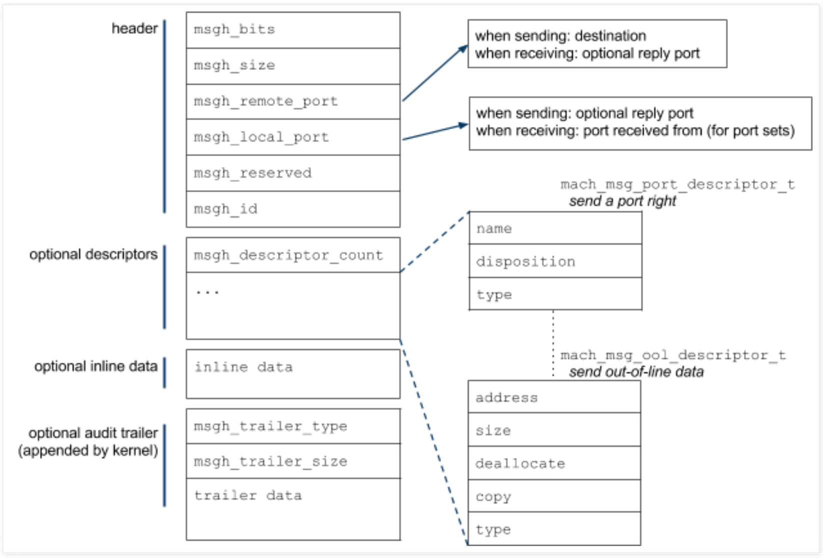

kernel-maintained mach message queues:
PID 123
port map {
0xabc : pointer to message queue
}
m->msgh_remote_port = 0xabc;
mach_msg(m, MACH_SEND_MSG...);
PID 456
port map {
0xdef : pointer to message queue
}
mach_msg(m, MACH_RCV_MSG...,
rcv_name=0xdef);
kernel
userspace

In this diagram we can see that the process with PID `123` has a mach port name `0xabc`. It's important to notice that this Mach port name only has a meaning within this process - we can see that in the kernel structure for this process `0xabc` is just a key which maps to a pointer to a message queue.

When the process with PID `456` tries to dequeue a message using the mach port name `0xdef` the kernel uses `0xdef` to index that process's map of mach ports such that it can find the correct message queue from which to dequeue a message.

Mach messages
A single mach message can have up to four parts:

- Message header - this header is mandatory and specifies the port name to send the message to as well as various flags.
- Kernel processed *descriptors* - this optional section can contain multiple descriptors which are parts of the message which need to be interpreted by the kernel.
- Inline data - this is the inline binary payload.
- Audit trailer - The message receiver can request that the kernel append an audit trailer to received messages.



When a simple mach message containing no descriptors is sent it will first be copied entirely into a kernel heap-allocated buffer in the kernel. A pointer to that copy is then appended to the correct mach message queue and when the process with a receive right to that queue dequeues that message the kernel copy of the message gets copied into the receiving process.

Out-of-line memory
Copying large messages into and out of the kernel is slow, especially if the messages are large. In order to send large amounts of data you can use the "out-of-line memory" descriptor. This enables the message sender to instruct the kernel to make a copy-on-write virtual memory copy of a buffer in the receiver process when the message is dequeued.

Bi-directional messages
Mach IPC is fundamentally uni-directional. In order to build a two-way IPC mechanism mach IPC allows for messages to carry port rights. In a mach message, along with binary data you can also send a mach port right.

Mach IPC is quite flexible when it comes to sending port rights to other processes. You can use the `local_port` field of the mach message header, use a port descriptor or use an OOL-ports descriptor. There are a multitude of flags to control exactly what rights should be transferred, or if new rights should be created during the send operation (it's common to use the `MAKE_SEND` flag which creates and sends a new send right to a port which you hold the receive right for.)

Bootstrapping Mach IPC
There's a fundamental bootstrapping problem with mach IPC: how do you get a send right to a port for which another process has a receive right without first sending them a message (thus encountering the same problem in reverse.)

One way around this could be to allow mach ports to be inherited across a `fork()` akin to setting up a pipe between a parent and child process using `socketpair()`. However, unlike file descriptors, mach port rights are not inherited across a fork so you can't implement such a system.

Except, some mach ports *are* inherited across a fork! These are the special mach ports, one of which is the bootstrap port. The parent of all processes on OS X is launchd, and one of its roles is to set the default bootstrap port which will then be inherited by every child.

Launchd
Launchd holds the receive-right to this bootstrap port and plays the role of the bootstrap server, allowing processes to advertise named send-rights which other processes can look up. These are OS X Mach IPC services.

MIG
We're now at the point where we can see how the kernel and userspace Mach IPC systems use a few hacks to get bootstrapped such that they're able to send binary data. This is all that you get with raw Mach IPC.

MIG is the Mach Interface Generator and it provides a simple RPC (remote procedure call) layer on top of the raw mach message IPC. MIG is used by all the Mach kernel services and many userspace services.

MIG interfaces are declared in `.defs` files. These use a simple Interface Definition Language which can define function prototypes and simple data structures. The MIG tool compiles the `.defs` into C code which implements all the required argument serialization/deserialization.

Calling a MIG RPC is completely transparent, it's just like calling a regular C function and if you've ever programed on a Mac you've almost certainly used a MIG generated header file.

IOKit
As mentioned earlier IOKit is the framework and kernel subsystem used for device drivers. All interactions with IOKit begin with the IOKit master port. This is another special mach port which allows access to the IOKit registry. `devices.defs` is the relevant MIG definition file. The Apple developer documentation describes the IOKit registry in great detail.

The IOKit registry allows userspace programs to find out about available hardware. Furthermore, device drivers can expose an interface to userspace by implementing a UserClient.

The main way which userspace actually interacts with an IOKit driver's UserClient is via the `io_connect_method` MIG RPC:

```
type io_scalar_inband64_t = array[*:16] of uint64_t;
type io_struct_inband_t   = array[*:4096] of char;

routine io_connect_method(
  connection            : io_connect_t;
  in    selector        : uint32_t;

  in    scalar_input    : io_scalar_inband64_t;
  in    inband_input    : io_struct_inband_t;
  in    ool_input       : mach_vm_address_t;
  in    ool_input_size  : mach_vm_size_t;

  out   inband_output   : io_struct_inband_t, CountInOut;
  out   scalar_output   : io_scalar_inband64_t, CountInOut;
  in    ool_output      : mach_vm_address_t;
  inout ool_output_size : mach_vm_size_t
);
```

This method is wrapped by the IOKitUser library function `IOConnectCallMethod`.

The kernel implementation of this MIG API is in `IOUserClient.cpp` in the function

`is_io_connect_method`:

```
    kern_return_t is_io_connect_method
     (
      io_connect_t connection,
      uint32_t selector,
      io_scalar_inband64_t scalar_input,
      mach_msg_type_number_t scalar_inputCnt,
      io_struct_inband_t inband_input,
      mach_msg_type_number_t inband_inputCnt,
      mach_vm_address_t ool_input,
      mach_vm_size_t ool_input_size,
      io_struct_inband_t inband_output,
      mach_msg_type_number_t *inband_outputCnt,
      io_scalar_inband64_t scalar_output,
      mach_msg_type_number_t *scalar_outputCnt,
      mach_vm_address_t ool_output,
      mach_vm_size_t *ool_output_size
      )
     {
         CHECK( IOUserClient, connection, client );

         IOExternalMethodArguments args;
...
         args.selector = selector;

         args.scalarInput = scalar_input;
         args.scalarInputCount = scalar_inputCnt;
         args.structureInput = inband_input;
         args.structureInputSize = inband_inputCnt;
...
         args.scalarOutput = scalar_output;
         args.scalarOutputCount = *scalar_outputCnt;
         args.structureOutput = inband_output;
         args.structureOutputSize = *inband_outputCnt;
...
         ret = client->externalMethod( selector, &args );
```

Here we can see that the code fills in an `IOExternalMethodArguments` structure from the arguments passed to the MIG RPC and then calls the `::externalMethod` method of the `IOUserClient`.

What happens next depends on the structure of the driver's `IOUserClient` subclass. If the driver overrides `externalMethod` then this calls straight into driver code. Typically the selector argument to `IOConnectCallMethod` would be used to determine what function to call, but if the subclass overrides `externalMethod` it's free to implement whatever method dispatch mechanism it wants. However if the driver subclass doesn't override `externalMethod` the `IOUserClient` implementation of it will call `getTargetAndMethodForIndex` passing the selector argument - this is the method which most `IOUserClient` subclasses override - it returns a pointer to an `IOExternalMethod` structure:

```
struct IOExternalMethod {
    IOService *  object;
    IOMethod     func;
    IOOptionBits flags;
    IOByteCount  count0;
    IOByteCount  count1;
};
```

Most drivers have a simple implementation of `getTargetAndMethodForType` which uses the `selector` argument to index an array of `IOExternalMethod` structures. This structure contains a pointer to the method to be invoked (and since this is C++ this isn't actually a function pointer but a pointer-to-member-method which means things can get very fun when you get to control it! See the bug report for CVE-2014-1379 in the Project Zero bugtracker for an example of this.)

The `flags` member is used to define what mixture of input and output types the `ExternalMethod` supports and the `count0` and `count1` fields define the number or size in bytes of the input and output arguments. There are various shim functions which make sure that `func` is called with the correct prototype depending on the declared number and type of arguments.

Putting all that together
At this point we know that when we call `IOConnectCallMethod` what really happens is that C code auto-generated by MIG serializes all the arguments into a data buffer which is wrapped in a mach message which is sent to a mach port we received received from the IOKit registry which we knew how to talk to because every process has a special device port. That message gets copied into the kernel where more MIG generated C code deserializes it and calls `is_io_connect_method` which calls the driver's `externalMethod` virtual method.

Writing an IOKit fuzzer

When auditing code alongside manual analysis it's often worth writing a fuzzer. As soon as you've understood where attacker-controlled data could enter a system you can write a simple piece of code to throw randomness at it. As your knowledge of the code improves you can make incremental improvements to the fuzzer, allowing it to explore the code more deeply.

`IOConnectCallMethod` is the perfect example of a API where this applies. It's very easy to write a simple fuzzer to make random `IOConnectCallMethod` calls. One approach to slightly improve on just using randomness is to try to mutate real data. In this case, we want to mutate valid arguments to `IOConnectCallMethod`. Check out this talk from Chen Xiaobo and Xu Hao about how to do exactly that.

DYLD interposing

dyld is the OS X dynamic linker. Similar to using LD_PRELOAD on linux dyld supports dynamic link time interposition of functions. This means we can intercept function calls between different libraries and inspect and modify arguments.

Here's the complete `IOConnectCallMethod` fuzzer interpose library I wrote for pwn4fun:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <IOKit/IOKitLib.h>

int maybe(){
  static int seeded = 0;
  if(!seeded){
    srand(time(NULL));
    seeded = 1;
  }
  return !(rand() % 100);
}

void flip_bit(void* buf, size_t len){
  if (!len)
    return;
  size_t offset = rand() % len;
  ((uint8_t*)buf)[offset] ^= (0x01 << (rand() % 8));
}

kern_return_t
fake_IOConnectCallMethod(
  mach_port_t connection,
  uint32_t    selector,
  uint64_t   *input,
  uint32_t    inputCnt,
  void       *inputStruct,
  size_t      inputStructCnt,
  uint64_t   *output,
  uint32_t   *outputCnt,
  void       *outputStruct,
  size_t     *outputStructCntP)
{
  if (maybe()){
    flip_bit(input, sizeof(*input) * inputCnt);
  }

  if (maybe()){
    flip_bit(inputStruct, inputStructCnt);
  }

  return IOConnectCallMethod(
    connection,
    selector,
    input,
    inputCnt,
    inputStruct,
    inputStructCnt,
    output,
    outputCnt,
    outputStruct,
    outputStructCntP);
}

typedef struct interposer {
  void* replacement;
  void* original;
} interpose_t;

__attribute__((used)) static const interpose_t interposers[]
  __attribute__((section("__DATA, __interpose"))) =
    {
      { .replacement = (void*)fake_IOConnectCallMethod,
        .original    = (void*)IOConnectCallMethod
      }
    };
```

Compile that as a dynamic library:

```
$ clang -Wall -dynamiclib -o flip.dylib flip.c -framework IOKit -arch i386 -
arch x86_64
```

and load it:

```
$ DYLD_INSERT_LIBRARIES=./flip.dylib hello_world
```

1% of the time this will flip one bit in any struct input and scalar input to an IOKit external method. This was the fuzzer which found the bug used to get kernel instruction pointer control for pwn4fun, and it found it well before I had any clue how the Intel GPU driver worked at all.

IntelAccelerator bug
Running the fuzzer shown above with any program using the GPU lead within seconds to a crash in the following method in the `AppleIntelHD4000Graphics` kernel extension at the instruction at offset `0x8BAF`:

```
IGAccelGLContext::unmap_user_memory(  ;rdi == this
    IntelGLUnmapUserMemoryIn *,       ;rsi
    unsigned long long)               ;rdx
__text:8AD6
__text:8AD6 var_30 = qword ptr -30h
...
__text:8AED  cmp   rdx, 8
__text:8AF1  jnz   loc_8BFB
__text:8AF7  mov   rbx, [rsi]        ;rsi points to controlled data
__text:8AFA  mov   [rbp+var_30], rbx ;rbx completely controlled
...
__text:8BAB  mov   rbx, [rbp+var_30]
__text:8BAF  mov   rax, [rbx]        ;crash
__text:8BB2  mov   rdi, rbx
__text:8BB5  call  qword ptr [rax+140h]
```

Looking at the cross references to this function in IDA Pro we can see that `unmap_user_memory` is selector `0x201` of the `IGAccelGLContent` user client. This external method has one struct input so on entry to this function `rsi` points to controlled data (and `rdx` contains the length of that struct input in bytes.)

At address `0x8af7` this function reads the first `8` bytes of the struct input as a qword and saves them in `rbx`. At this point `rbx` is completely controlled. This controlled value is then saved into the local variable `var_30`. Later at `0x8bab` this value is read back into `rbx`, then at `0x8baf` that controlled value is dereferenced without any checks leading to a crash. If that dereferences doesn't crash however, then the qword value at offset `0x140` from the read value will be called.

In other words, this external method is treating the struct input bytes as containing a pointer to a C++ object and it's calling a virtual method of that object without checking whether the pointer is valid. Kernel space is just trusting that userspace will only ever pass a valid kernel object pointer. So by crafting a fake IOKit object and passing a pointer to it as the struct input of selector 0x201 of `IGAccelGLContent` we can get kernel instruction pointer control! Now what?

SMEP/SMAP
SMEP and SMAP are two CPU features designed to make exploitation of this type of bug trickier.

Mavericks supports Supervisor Mode Execute Prevention which means that when the processor is executing kernel code the cpu will fault if it tries to execute code on pages belonging to userspace. This prevents us from simply mapping an executable kernel shellcode payload at a known address in userspace and getting the kernel to jump to it.

The generic defeat for this mitigation is code-reuse (ROP). Rather than diverting execution directly to shellcode in userspace instead we have to divert it to existing executable code in the kernel. By "pivoting" the stack pointer to controlled data we can easily chain together multiple code chunks and either turn off SMEP or execute an entire payload just in ROP.

The second generic mitigation supported at the CPU level is Supervisor Mode Access Prevention. As the name suggests this prevents kernel code from even reading user pages directly. This would mean we'd have to be able to get controlled data at a known location in kernel space for the fake IOKit object and the ROP stack since we wouldn't be able to dereference userspace addresses, even to read them.

However, Mavericks doesn't support SMAP so this isn't a problem, we can put the fake IOKit object, vtable and ROP stack in userspace.

kASLR
To write the ROP stack we need to know the exact location of the kernel code we're planning to reuse. On OS X kernel address space layout randomisation means that there are 256 different addresses where

the kernel code could be located, one of which is randomly chosen at boot time. Therefore to find the addresses of the executable code chunks we need some way to determine the distance kASLR has shifted the code in memory (this value is known as the kASLR slide.)

IOKit registry

We briefly mentioned earlier that the IOKit registry allows userspace programs to find out about hardware, but what does that actually mean? The IOKit registry is really just a place where drivers can publish (key:value) pairs (where the key is a string and the value something equivalent to a CoreFoundation data type.) The drivers can also specify that some of these keys are configurable which means userspace can use the IOKit registry API to set new values.

Here are the MIG RPCs for reading and settings IOKit registry values:

```
routine io_registry_entry_get_property(
          registry_entry      : io_object_t;
      in  property_name       : io_name_t;
      out properties          : io_buf_ptr_t, physicalcopy );

routine io_registry_entry_set_properties(
          registry_entry      : io_object_t;
      in  properties          : io_buf_ptr_t, physicalcopy;
      out result              : kern_return_t );
```

And here are the important parts of the kernel-side implementation of those functions, firstly, for setting a property:

```
kern_return_t is_io_registry_entry_set_properties(
    io_object_t registry_entry,
    io_buf_ptr_t properties,
    mach_msg_type_number_t propertiesCnt,
    kern_return_t * result){
  ...

  obj = OSUnserializeXML( (const char *) data, propertiesCnt );
  ...
#if CONFIG_MACF
  else if (0 != mac_iokit_check_set_properties(kauth_cred_get(),
                                                 registry_entry,
                                                 obj))
    res = kIOReturnNotPermitted;
#endif
  else
    res = entry->setProperties( obj );
  ...
```

and secondly, for reading a property:

```
kern_return_t is_io_registry_entry_get_property(
    io_object_t registry_entry,
    io_name_t property_name,
    io_buf_ptr_t *properties,
    mach_msg_type_number_t *propertiesCnt ){
  ...
  obj = entry->copyProperty(property_name);
  if( !obj)
    return( kIOReturnNotFound );

  OSSerialize * s = OSSerialize::withCapacity(4096);
  ...
  if( obj->serialize( s )) {
    len = s->getLength();
    *propertiesCnt = len;
    err = copyoutkdata( s->text(), len, properties );
  ...
```

These functions are pretty simple wrappers around the `setProperties` and `copyProperty` functions implemented by the drivers themselves.

There's one very important thing to pick up on here though: in the `is_io_registry_entry_set_properties` function there's a MAC hook, highlighted here in red, which allows sandbox profiles to restrict the ability to set IOKit registry values. (This hook is exposed by Sandbox.kext as the `iokit-set-properties` operation.) Contrasts this with the `is_io_registry_entry_get_property` function which has no MAC hook. This means that read access to the IOKit registry cannot be restricted. *Every* OS X process has full access to read every single (key:value) pair exposed by every IOKit driver.

Enumerating the iokit registry

OS X ships with the `ioreg` tool for exploring the IOKit registry on the command line. By passing the `-l` flag we can get `ioreg` to enumerate all the registry keys and dump their values. Since we're looking for kernel pointers, lets grep the output looking for a byte pattern we'd expect to see in a kernel pointer:

```
$ ioreg -l | grep 80ffffff
    |   "IOPlatformArgs" = 
<00901d2880ffffff00c01c2880ffffff90fb222880ffffff0000000000000000>
```

That looks an awful lot like a hexdump of some kernel pointers :)

Looking for the "`IOPlatformArgs`" string in the XNU source code we can see that the first of these pointers is actually the address of the `DeviceTree` that's passed to the kernel at boot. And it just so happens that the same kASLR slide that gets applied to the kernel image also gets applied to that `DeviceTree` pointer, meaning that we can simply subtract a constant from this leaked pointer to determine the runtime load address of the kernel allowing us to rebase our ROP stack.

Check out this blog post from winocm for a lot more insight into this bug and its applicability to iOS.

OS X kernel ROP pivot
Looking at the disassembly of `unmap_user_memory` we can see that when the controlled virtual method is called the `rax` register points to the fake vtable which we've put in userspace. The pointer at offset `0x140h` will be the function pointer that gets called which makes the vtable a convenient place for the ROP stack. We just need to find a sequence of instructions which will move the value of `rax` into `rsp`. The `/mach_kernel` binary has following instruction sequence:

```
    push rax
    add [rax], eax
    add [rbx+0x41], bl
    pop rsp
    pop r14
    pop r15
    pop rbp
    ret
```

This will push the vtable address on to the stack, corrupt the first entry in the vtable and write a byte to `rbx+0x41`. `rbx` will be the this pointer of the fake IOKit object which we control and have pointed into userspace so neither of these writes will crash. `pop rsp` then pops the top of the stack into `rsp` - since we just pushed `rax` on to the stack this means that `rsp` now points to the fake vtable in userspace. The code then pops values for `r14`, `r15` and `rbp` then returns meaning that we can place a full ROP stack in the fake vtable of the fake IOKit object.

Payload and continuation
The OS X kernel function `KUNCExecute` is a really easy way to launch GUI applications from kernel code:

```
kern_return_t KUNCExecute(char executionPath[1024], int uid, int gid)
```

The payload for the pwn4fun exploit was a ROP stack which called this, passing a pointer to the string `"/Applications/Calculator.app/Contents/MacOS/Calculator"` as the `executionPath` and `0` and `0` as the `uid` and `gid` parameters. This launches the OS X calculator as root :-)

Take a look at this exploit for this other IOKit bug which takes a slightly different approach by using a handful of ROP gadgets to first disable SMEP then call a more complicated shellcode payload in userspace. And if you're still running OS X Mavericks or below then why not try it out?

After executing the kernel payload we can call the kernel function `thread_exception_return` to return back to usermode. If we just do this however it will appear as if the whole system has frozen. The kernel payload has actually run (and we can verify this by attaching a kernel debugger) but we can no longer interact with the system. This is because before we got kernel code execution `unmap_user_memory` took two locks - if we don't drop those locks then no other functions will be able to get them and the GPU driver grinds to a halt. Again, check out that linked exploit above to see some example shellcode which drops the locks.

Conclusion
The actual development process of this sandbox escape was nothing like as linear as this writeup made it seem. There were many missed turns and other bugs which looked like far too much effort to exploit. Naturally these were reported to Apple too, just in case.

A few months after the conclusion of pwn4fun 2014 I decided to take another look at GPU drivers on OS X, this time focusing on manual analysis. Take a look at the following bug reports for PoC code and details of all the individual bugs: CVE-2014-1372, CVE-2014-1373, CVE-2014-1376, CVE-2014-1377, CVE-2014-1379, CVE-2014-4394, CVE-2014-4395, CVE-2014-4398, CVE-2014-4401, CVE-2014-4396, CVE-2014-4397, CVE-2014-4400, CVE-2014-4399, CVE-2014-4416, CVE-2014-4376, CVE-2014-4402

Finally, why not subscribe to the Project Zero bug tracker and follow along with all our latest research?

Posted by Unknown at 3:52 PM

1 comment:

nuncupatory December 1, 2014 at 4:24 AM

Intricate and absolutely fascinating! I'd love to hear about the "wrong turns" and dead-ends you encountered before you identified this.

Also, as far as I'm aware, SMAP will only arrive with Broadwell processors, it evidently missed Haswell.
Keep up the good work!
Derek

Reply

Add comment

Enter your comment...

Comment as: Sign out

Publish Preview Notify me