Visualizing Page Tables for Exploitation

by Alexandru Radocea and Georg Wicherski

1. Introduction

Modern processors feature the distinction between physical memory address space and virtual address space. While the physical address space references the actual memory cells on the RAM, the virtual address space is an alias space that is mapped to physical address space using page tables. This allows a program that utilizes a hardcoded memory address *X* to be run in two instances simultaneously by mapping the two instances' respective addresses *X* to X_1 and X_2 in the physical address space.

Besides fulfilling these simple mappings, modern processors' page tables encode various memory protections that are checked upon accessing a virtual memory address. The ARM and x86_64 processors we are covered with this research support the following noteworthy memory protections:

- Pages not present at all. Any access to these virtual memory locations results in an error.
- Read-only versus read-write protection. If a read-only location is attempted to be written, an error is generated.
- Executable memory versus non-executable (data only) memory. It is an error if the instruction pointer points to a virtual address within a page explicitly marked as not executable.
- The privilege level required to access a page, allowing certain pages to be accessible only by the running operating system kernel. While most operating systems define a virtual address range that is reserved for the user-space (i.e. per default < `0x80000000` on Windows and < `0xc0000000` on Linux), pages outside the range can still be readable per their page protections.

An attacker can now try to find pages with a specific selection of the aforementioned permissions that do not represent a required set of minimal permissions well. One example might be any pages that are marked both writable and executable, as they might allow an attacker to directly place native code in those memory areas and divert code execution there. Another example would be a page containing sensitive kernel information that is (at least for reading) accessible to user-space.

Besides looking for pages with permissions favorable to an attacker, specific pages that exhibit a static virtual address repeatedly are of interest because they are indicative of weak or incomplete ASLR implementations.

In this work, we try to motivate different visualizations of page tables for finding these flaws easily and in a manner easy to convey. Instead of collecting our data from operating system interfaces, we directly queried the hardware specific page tables to overcome any representation bias the kernel might introduce (such as not providing specific driver mappings that the kernel API is unaware about).

2. Data Collection Methodology

The page tables to visualize were all collected from real devices in various ways. On the hardware architectures examined, reading page tables requires accessing physical memory and reading paging configuration hardware registers–both only accessible from privileged kernel-space. This section explains how access to this privileged information was achieved on the respective platforms and how the data was subsequently collected.

iOS 6.x on ARMv7 (iPhone 4, A4, Cortex-A8)

The iOS operating system provides no accommodations for third-party kernel development. For many iOS releases now, the user-land address space has been heavily constrained to maximize security properties. With the exception of JIT pages and barring any implementation bugs, applications cannot dynamically map code. Every page of executable code must have a matching hash which is code signed, and the XNU operating system enforces code signing of every process. Exploits must use return oriented programming and cannot ever introduce new code without kernel privilege escalation.

Similarly, the XNU kernel on iOS is locked down to prevent third parties from loading modules or kernel extensions. Even with root privileges, it is not possible to load new kernel code as the XNU kext loading code is not compiled in. And the kernelcache, which is loaded by iBoot, is codesigned using public key cryptography chained to Apple certificates, further preventing arbitrary kernel code injection.

To get access to physical memory pages and paging control registers, a security weakness in a jailbroken iOS device was leveraged to inject and run supervisor code.

Jailbreak assisted kernel programming

Jailbreaks such as evasi0n and redn0w patch the XNU kernel to re-enable debugging of the kernel task with the `task_for_pid` mach routine. This routine allows for acquiring a task port that can be used read and write arbitrary memory to the kernel task, as well as enumeration of virtual memory data structures.

Since iOS disables kernel extension loading in XNU, no dynamic linker is available. To work around this, driver code is written as position independent code and external functions in XNU are manually discovered through reverse engineering of the kernel cache.

Three new system calls are introduced. Two transfer virtual and physical memory using the IOKit subsystem in XNU. The third transfers relevant ARM paging control registers such as `ttr0`, `ttbr1`, `ttbcr`, `sctrl`, and `dacr`. Driver code was also written to enumerate all running processes' physical mappings by walking the linked list of all processes.

Jailbreak limitations

The *evasi0n* jailbreak was used during testing. It was discovered that the exploit modifies page tables to mark the entire kernel cache as writable. To acquire accurate data, the evasi0n jailbreak was modified to run the page table dumping code before "untethered jailbreaking" completes**.** This allows for collecting page table information before eavsi0n modifies the kernel mappings.

Android on ARMv7-A

Data for the Android operating system that is based on the Linux kernel has been collected using two different methodologies. For the Nexus 4 device that allows flashing a custom kernel easily and has a well-documented kernel build process, a kernel patch that provides access to the required hardware registers has been developed. This allows not only capturing data in a legitimate way but also capturing all page tables for all processes that are currently not scheduled and therefore not represented with their page tables in the respective hardware registers.

Because not all Android devices are equally open, we also wrote a local kernel exploit based collection utility. This allows collecting data from arbitrary kernels without relying on flashing any custom kernel patches to the device.

Custom kernel patch based collection

The Nexus 4 phone runs a Linux 3.4 derived Android kernel with the source readily available at `https://android.googlesource.com/kernel/msm` in the `android-msm-mako-3.4-jb-mr1.1` branch. Instructions for building and flashing the kernel are available from the Android documentation. Physical memory is already accessible via the `/dev/mem` device that provides a file-like API for reading and writing physical memory. This memory device has been extended with a kernel patch to support an ioctl for fetching the values of the ttbcr, ttbr0, ttbr1 and sctlr hardware registers. Additionally, the patch allows iterating over all registered processes and fetching their respective ttbr0 values that will be set on a context switch to that process. The kernel patch against the official Android Nexus 4 tree is included in our code release.

After booting into the patched kernel, a user-space program running with root privileges (or more accurately with the `CAP_SYS_RAWIO` capability) can then open the `/dev/mem` device, query the hardware registers and dump the page tables.

Local kernel exploit based collection

To obtain the `CAP_SYS_RAWIO` capability (required to read the page tables from `/dev/mem`) and to read the configuration registers' values, we used a modified CVE-2013-2094 exploit. We then used the same dumping procedure as for dumping with a legitimate kernel patch, albeit limited to the current process' ttbr0 for reasons of portability.

XNU on x86_64

This data was collected for OS X running Mountain Lion (10.8).

A kernel extension was written that can dump paging control registers to user-land as well as transfer physical memory. A user-land driver program uses the driver to dump control registers and enumerate page tables in physical memory. In addition, the listing of all running processes is walked to dump page tables to retrieve and enumerate the physical mapping for every running process.

3. Visualization Methodology

As explained in the introduction, we chose different visualization methods to visualize the following page table entry properties:

- Permissions
    - Writable and Executable

    - Accessibility for user-space
- Virtual address
- Physical address
- Page size

Hierarchical Visualization

Hierarchical views show how the address space is logically divided, for example how regions such as shared libraries are loaded and aliased, how drivers allocate DMA memory, how virtual aliases to all of physical memory are set up.

With x86-64 ia32.e paging, there are 3 layers of indirection (four layers counting the initial PML4 entry) for a page translation. With ARMv7 in short-descriptor mode, there are just two.

While hierarchical visualization is great for educational purposes and it reflects the CPU logic traversing different tables well, it makes it hard to get an overview about security properties. Security properties can be inherited across hierarchical layers on some architectures but often can also be defined at the lowest level. Drilling down to the lowest level is therefore necessary and the purposed of the hierarchical visualization is hence nullified. Therefore, we focus on other visualization methods in this work.

Hilbert Curves

Linear mappings of page table properties become difficult to identify visually. When graphed linearly, the different regions can become narrow bands that are difficult to see visually. By applying Hilbert Curves to mappings, regions are spatially translated into clearly identifiable spaces.

One challenge when visualizing page tables that way is how to deal with the lack of data, or sparseness in the address space. For example, the IA32.e address space spans 48 bits, or 128 terabytes. This is vastly more address space than there is RAM available with today's technology. So when visualizing the entire possible address space, the gaps between mappings can be shrunken and approximated over a maximum gap size to deemphasize the lack of information and provide for visually meaningful images.

ASLR Heatmap

Heatmaps of real, observed ASLR offsets can show practical weaknesses in ASLR implementations. These heatmaps are derived from the Hilbert Curve visualization. Since there are multiple slides and things moving around, looking at things from the page table layer will quickly show everything that is—or is not—well randomized for both user-land processes and kernel components.

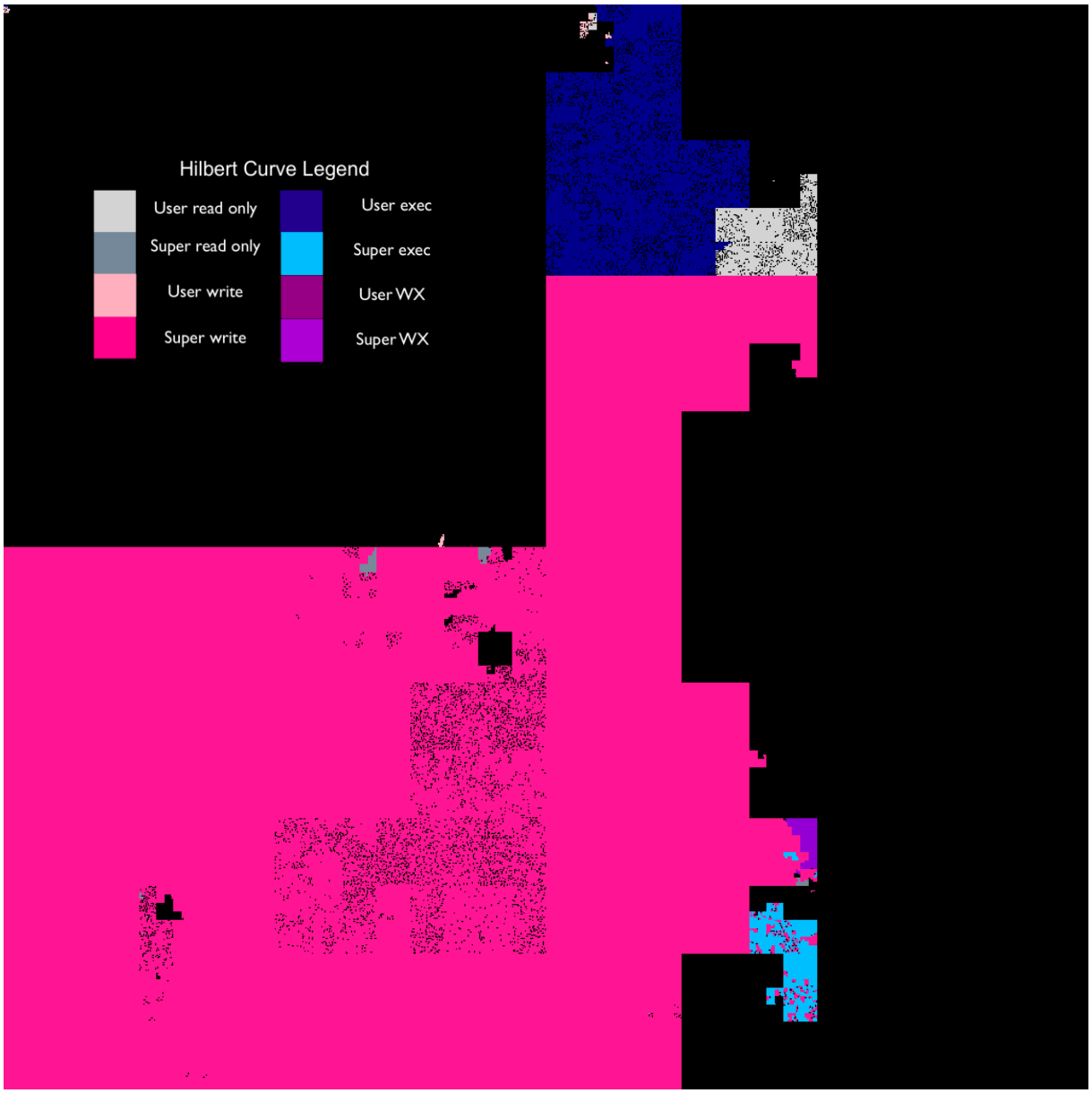
4. Case Studies and Findings

This section explains the various security deficiencies we found applying the visualizations to the different targets.

XNU on x86_64

The OS X operating system runs the XNU kernel. Starting with OS X 10.8, the kernel and kernel extensions run exclusively in 64-bit mode on Intel processors. The paging mechanism is IA32e/AMD64, and can map 48-bits of addressable memory using a four-level page table hierarchy.

Visualization yields a number of interesting characteristics: kernel ASLR (randomized per-boot), W^X kernel code, a fixed shared cache across userspace processes (per-boot), randomized *dyld* and preemption-free zone (per-execution), as well as randomized executable text (per-execution when linked with ASLR support), and some writable regions of code still in the kernel mappings:



OS X 10.8 kernel and user space

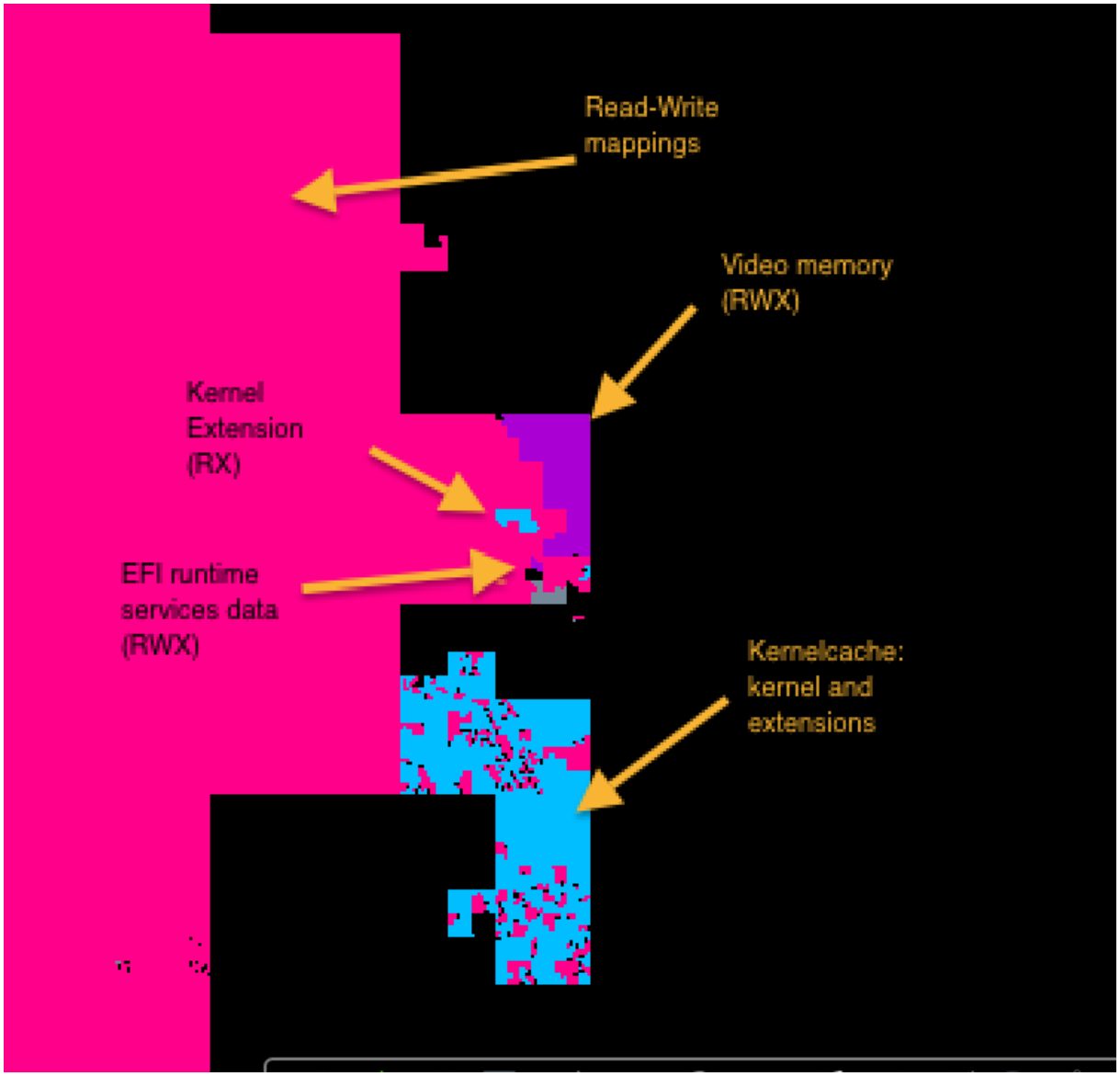
Kernel ASLR

The kernel stack and heap as well as the kernel code are randomized (kernel cache including the kernel and extensions as well as additional kernel extensions). These are randomized separately.

Physical mappings are also randomized, so that consecutive boots will have kernel data and code located on unpredictable, distinct physical memory pages.

W^X kernel memory protections

Predictable, writeable, executable pages would make KASLR ineffecitve.



Visualization shows two randomized (non-predictable) regions mapped as writeable and executable. The first of these regions maps device memory, and corresponds to video memory. It is not actually possible to fetch and execute code from these pages. The second region, however, corresponds to EFI data and it is possible to execute code from this region after writing to it.

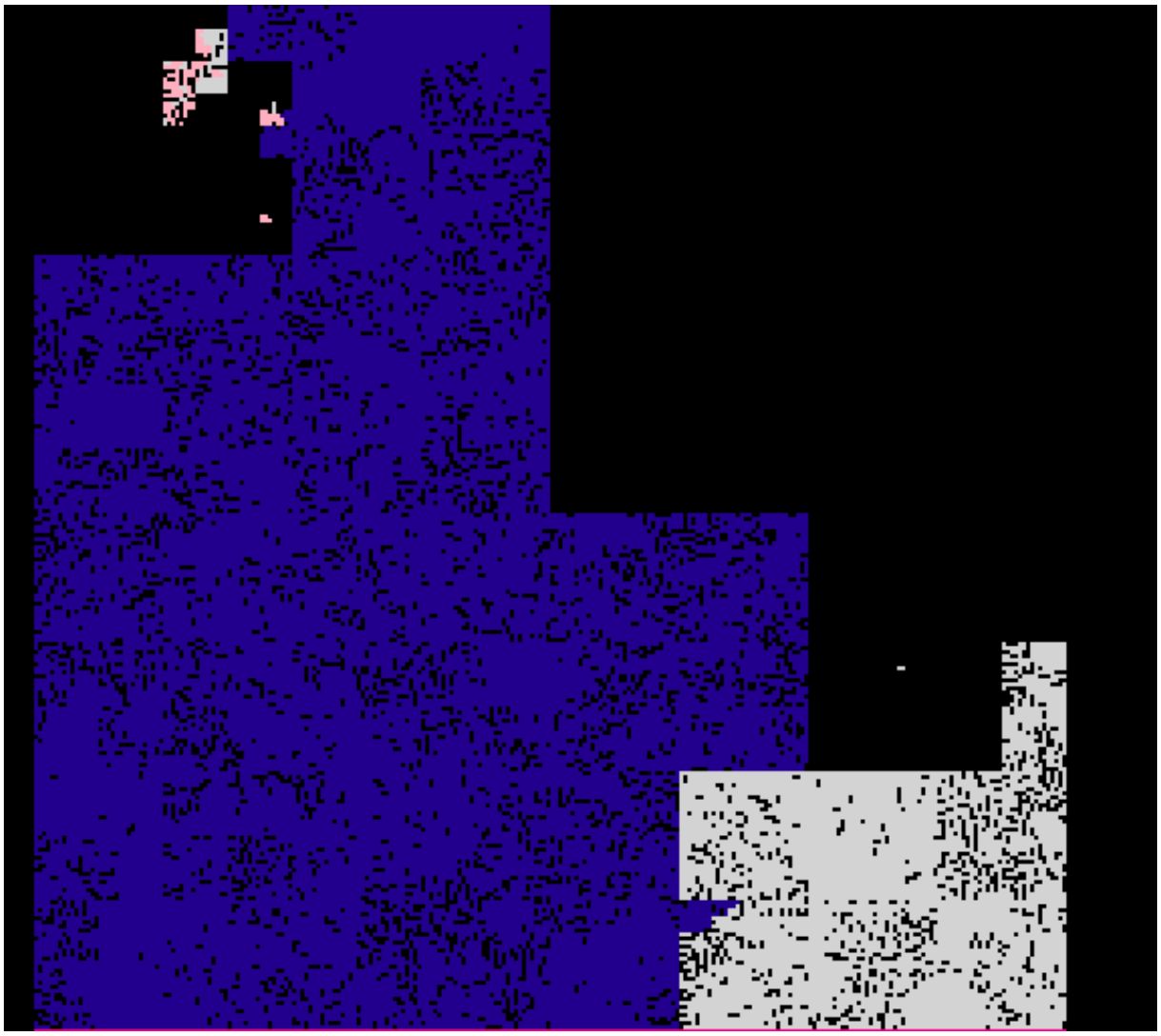
Both of these regions are randomized (in both virtual and physical space), and independently from other data in the address space, making them difficult to leverage in a kernel memory trespass exploit.

To inject code an attacker should load a kernel extension or leverage a gadget to turn off permission checking in paging control registers to write and execute injected code.

SMEP

Mountain Lion can use the Intel SMEP feature to prevent branching into user pages for kernel code execution.

Shared Cache



iOS 6.x on ARMv7-A

Starting with iOS 6.0, a number of important security improvements were introduced by Apple to create serious obstacles to kernel exploitation. Three are relevant to address space protection and pagetable layouts and discussed under findings.

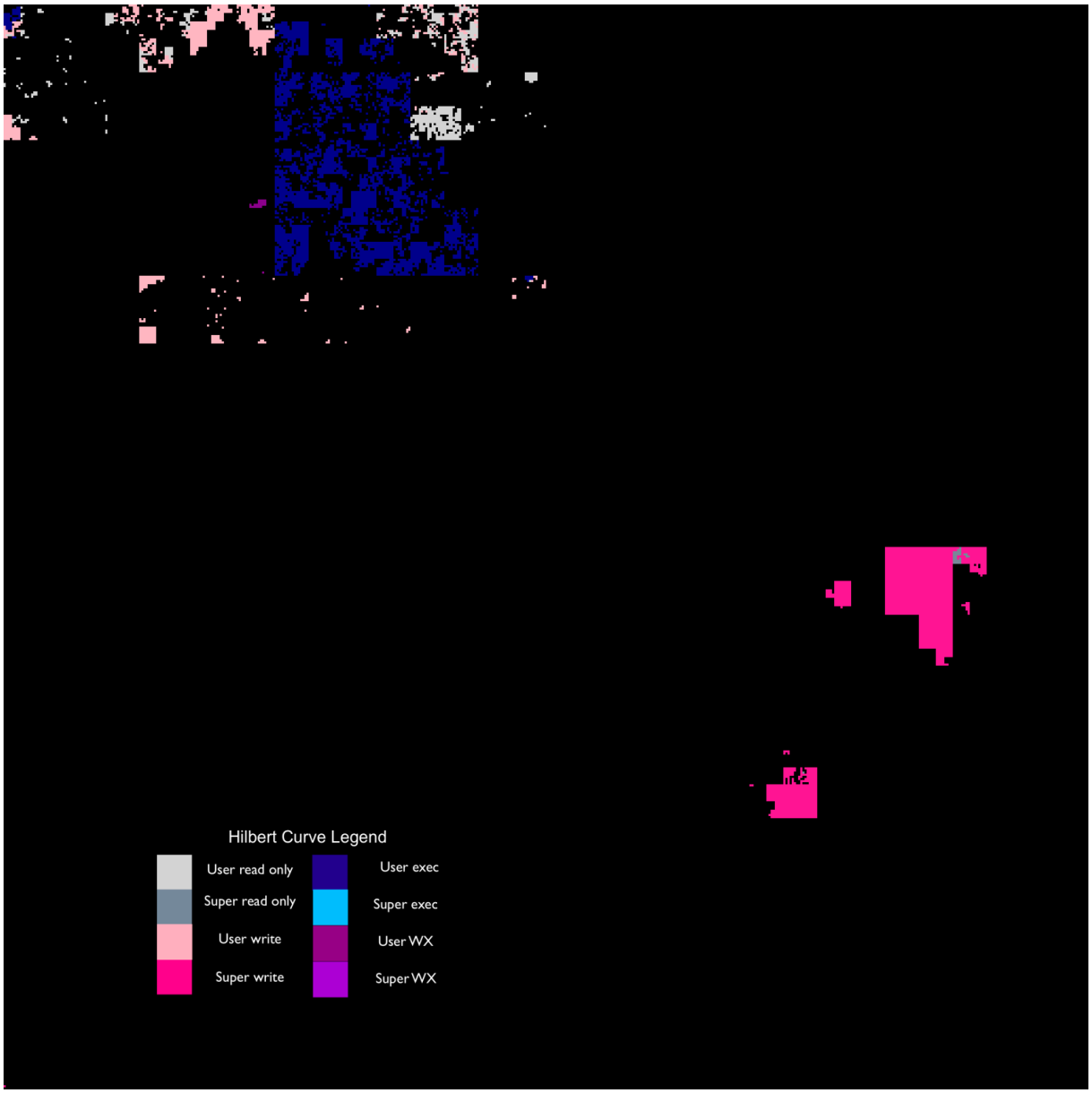
i. The kernel address space is tightened to make all kernel code as well as data structures, such as the system call table, read-only. Previously, the entire kernel cache region was marked writable meaning an exploit could easily patch code. Similarly, all data is now marked non executable, meaning a kernel exploit likely must use kernel return oriented programming to execute code.

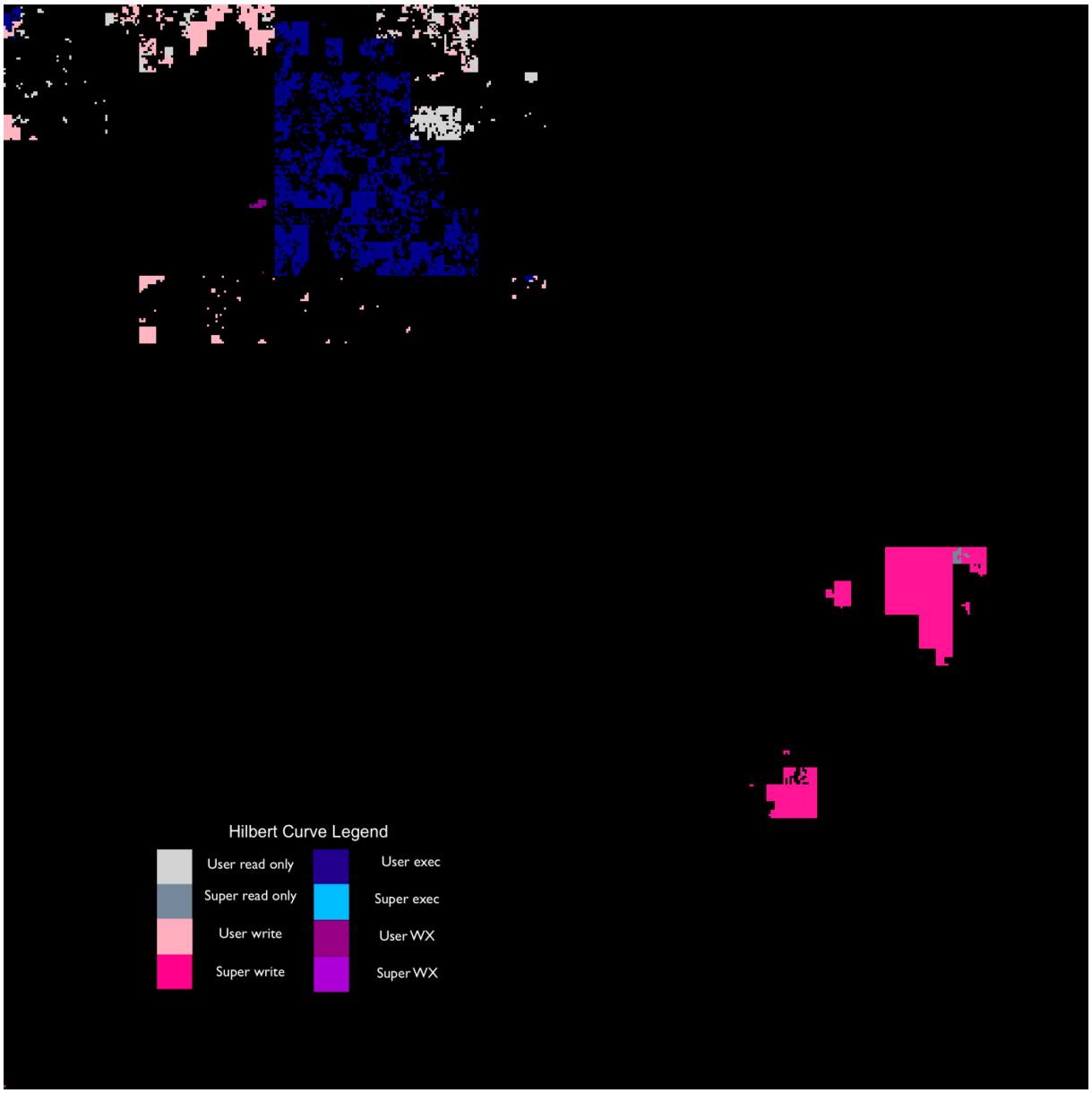
ii. Kernel Address Space Layout Randomization (KASLR) was introduced to make memory trespass errors harder to exploit. Two offsets are used: one arranged by iBoot to shift the location of the entire kernel cache, and a second by XNU to shift where the heap and stacks

will be located. KASLR, combined with tightened address space permissions, makes kernel return oriented programming much more unreliable without a good information leak.

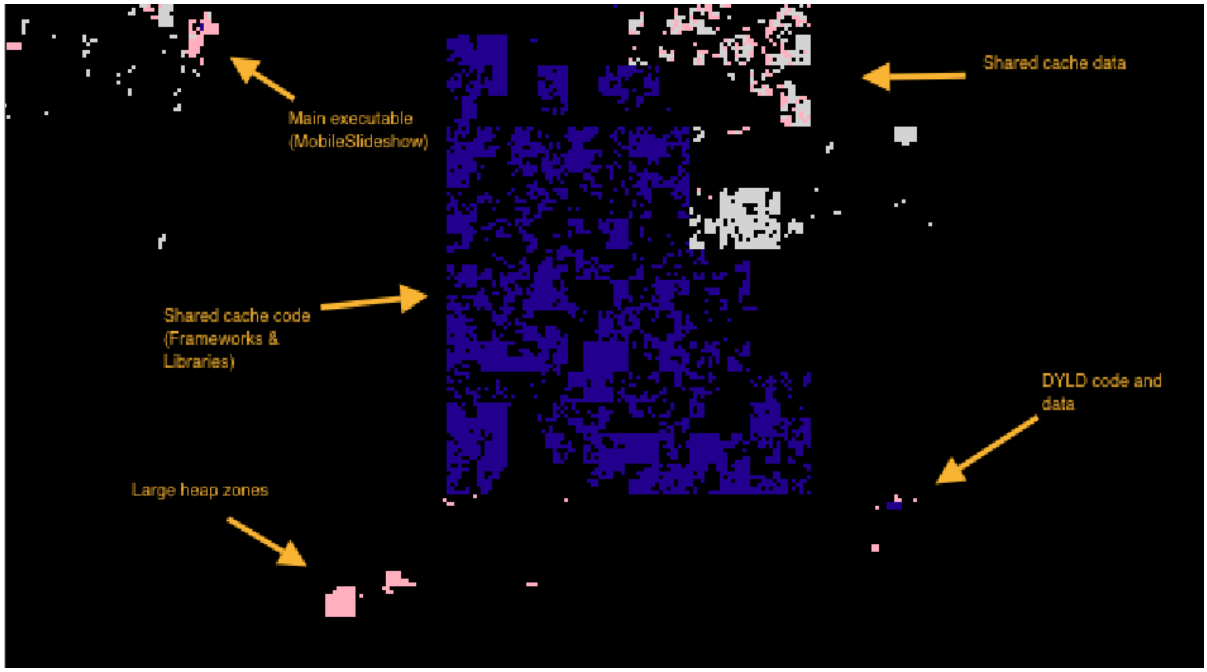
iii. User-space dereferences are blocked from kernel mode to prevent exploits from leveraging shared address space data and code. This was implemented by swapping paging base registers upon a context switch. More specifically, `ttbr0` is swapped to point from the current process user-land address space to the kernel map (matching `ttbr1`). This prevents address space sharing until a *copyin* or *copyout* routine temporarily swaps `ttbr0` back to reopen the user-land context's virtual address space for efficient data copying. With this security mechanism, kernel exploits are forced to also inject their ROP chains and other metadata into kernel space.

Sample programs: MobileSafari, MobileSlideshow

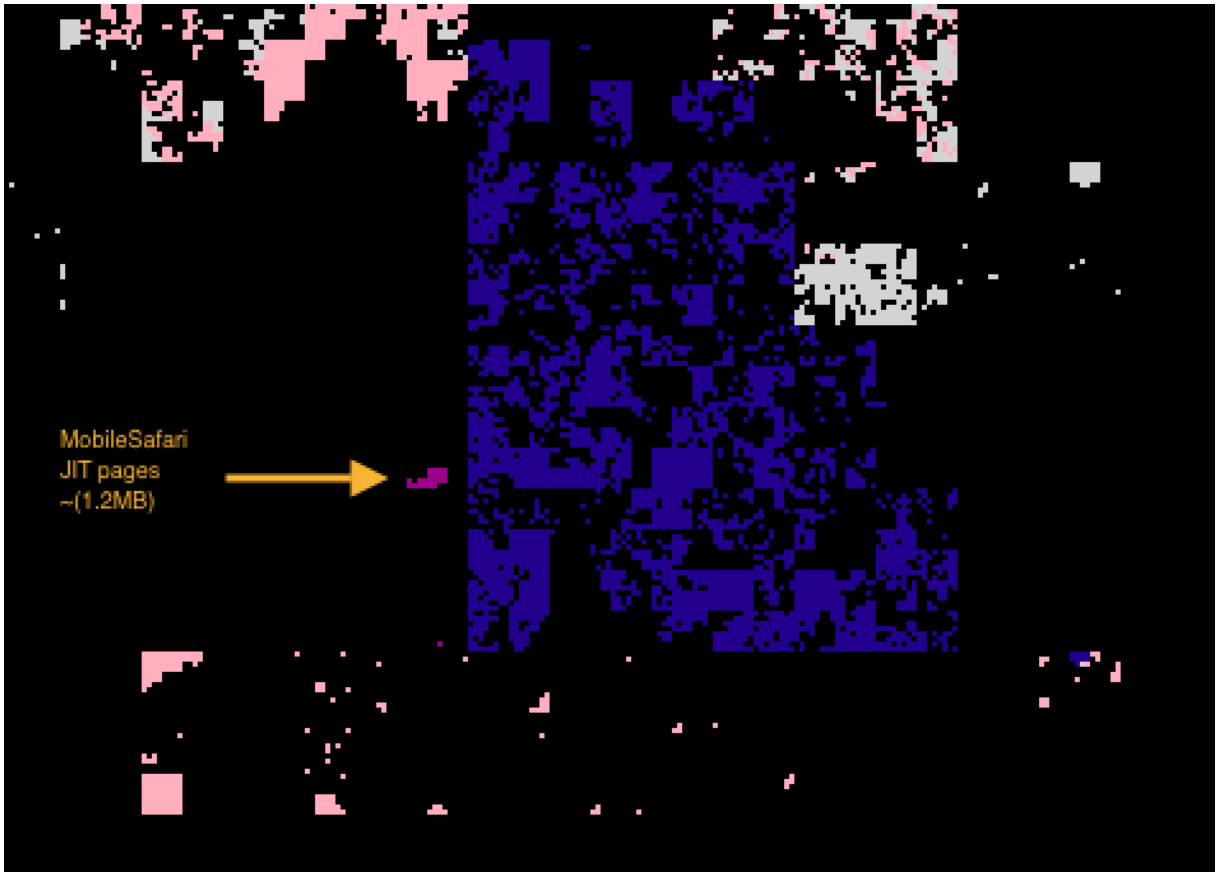



Hilbert Curve Legend
User read only
Super read only
User write
Super write
User exec
Super exec
User WX
Super WX

Annotated MobileSlideshow



MobileSafari JIT



MobileSafari employs a ~1.2 MB RWX memory region for Just In Time compilation of javascript code.

`

Fixed physical memory translations

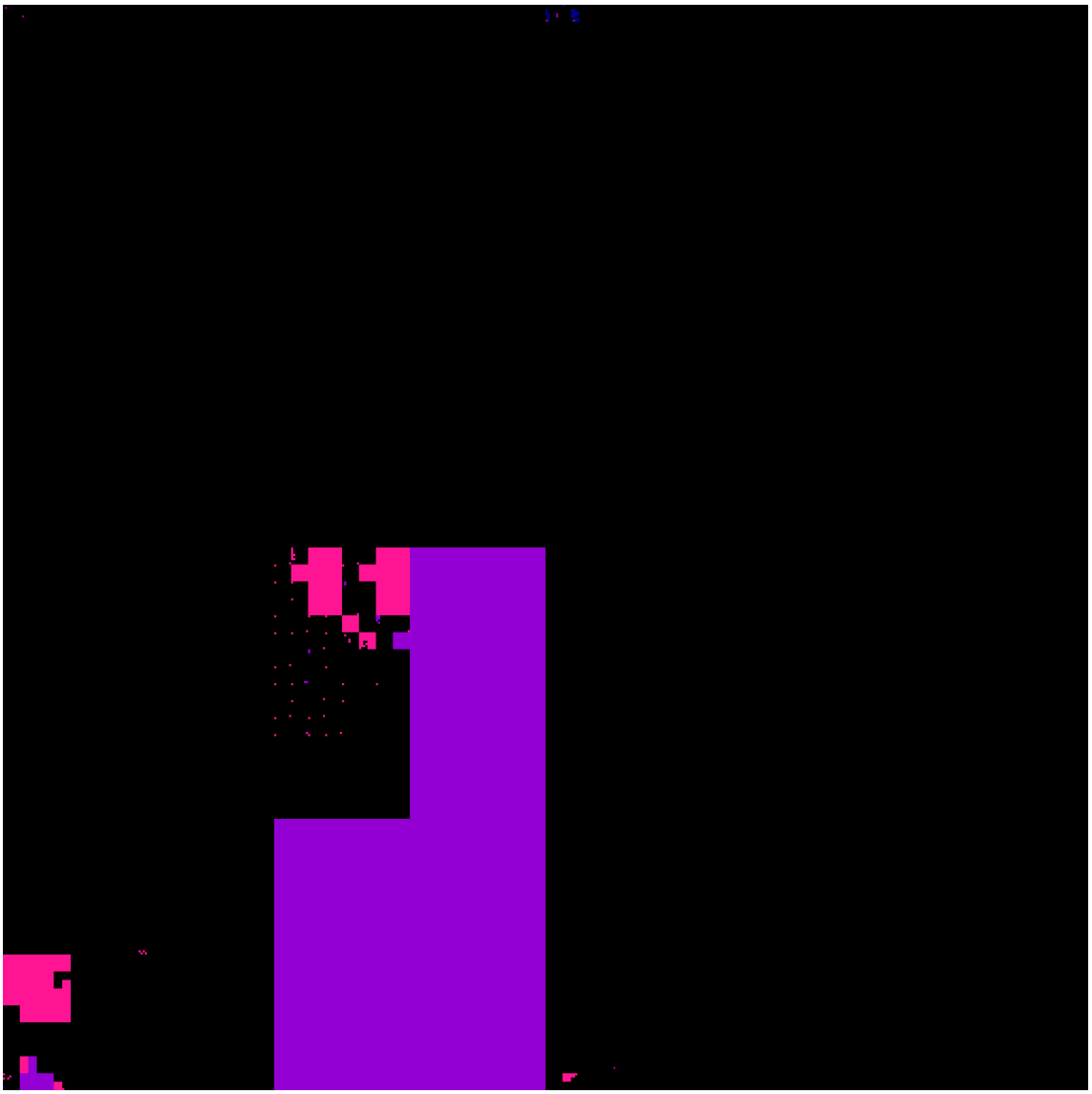
Physical memory mappings are not randomized. If a vulnerability leads to direct physical memory writes, or the kernel to virtual mapping can be leaked, a kernel exploit could reliably attack page table entries in physical memory to inject payloads.

Android on ARMv7-A

This section discusses the findings we made on different kernels running the latest Android 4.2.2 on various devices (including Nexus 4, Nexus 7, Galaxy Nexus and Galaxy S4). All findings in this section have been reported responsibly to Google, ARM, the Linux kernel security team, the Linaro security team and Qualcomm. Patches are anticipated for Linux 3.11.

Writable and Executable Kernel .text

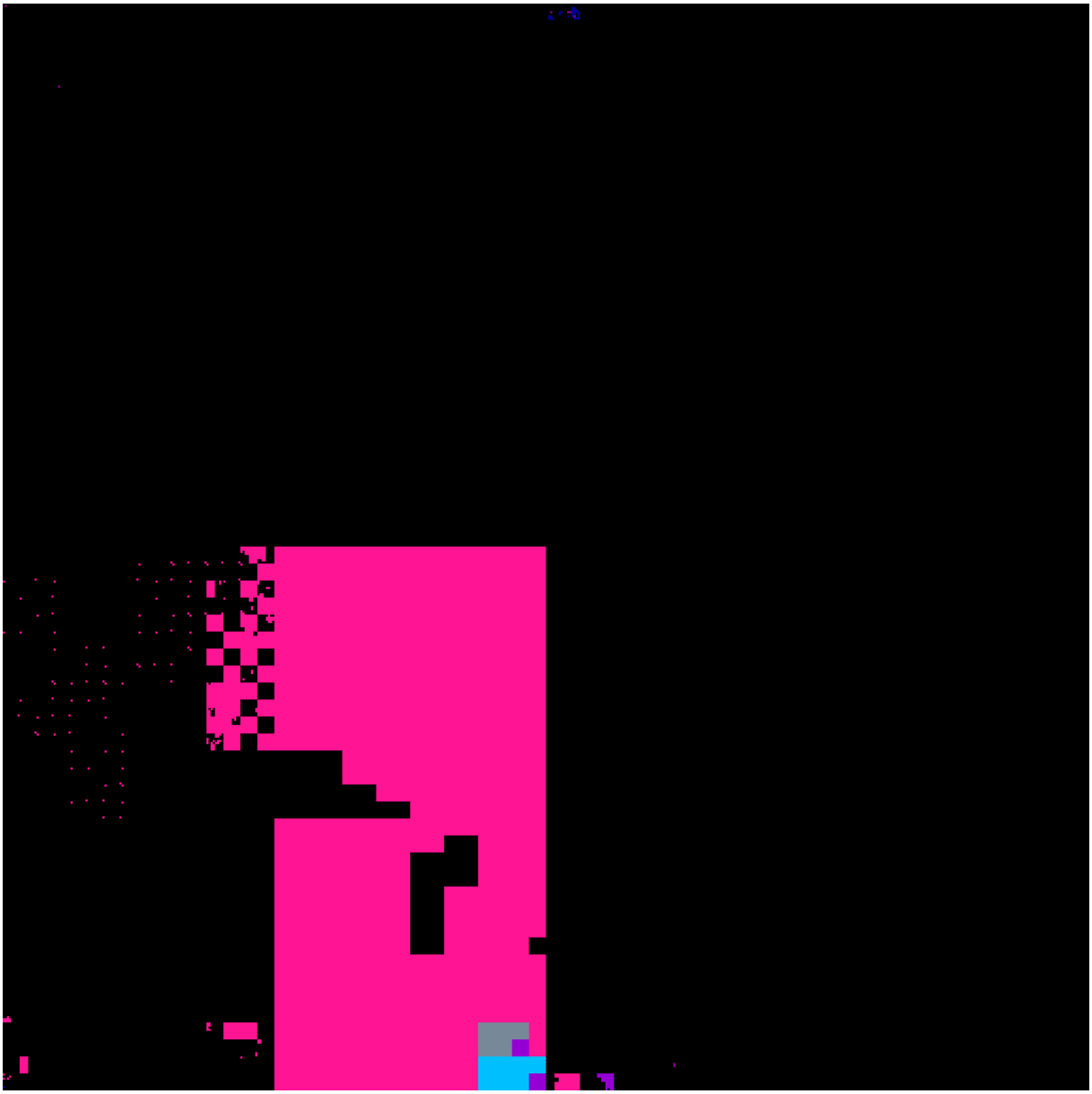
The immediately obvious finding on Linux on ARM is that for all inspected devices running any kernel but the Qualcomm MSM kernel, the entire kernel `.text` section is both writable and executable. Since the kernel's `.text` section is located at a predictable virtual and physical address, an attacker can use any writing primitive to directly replace kernel code with his native code. See below for an information leak of the system call handler function's address.



Executable and Writable Kernel .text on Nexus 7, Android 4.2.2

Looking at the kernel configuration, this is not unexpected behavior: `CONFIG_DEBUG_RODATA` is not activated on any of the kernel device configurations. The fact that making the kernel `.text` non-writable is a debug setting in the *Kernel Hacking* section speaks well for the state of security in Linux on ARM. It is questionable that vendors do not simply activate this setting by default, considering that at least the Nexus 4 stock kernel runs fine with this setting is enabled.

The MSM kernel mitigates this by another configuration setting that has been added to that specific source tree: `CONFIG_STRICT_MEMORY_RWX` makes most of the kernel code non-writable. Most data memory is marked non-executable just like in the mainline kernel.



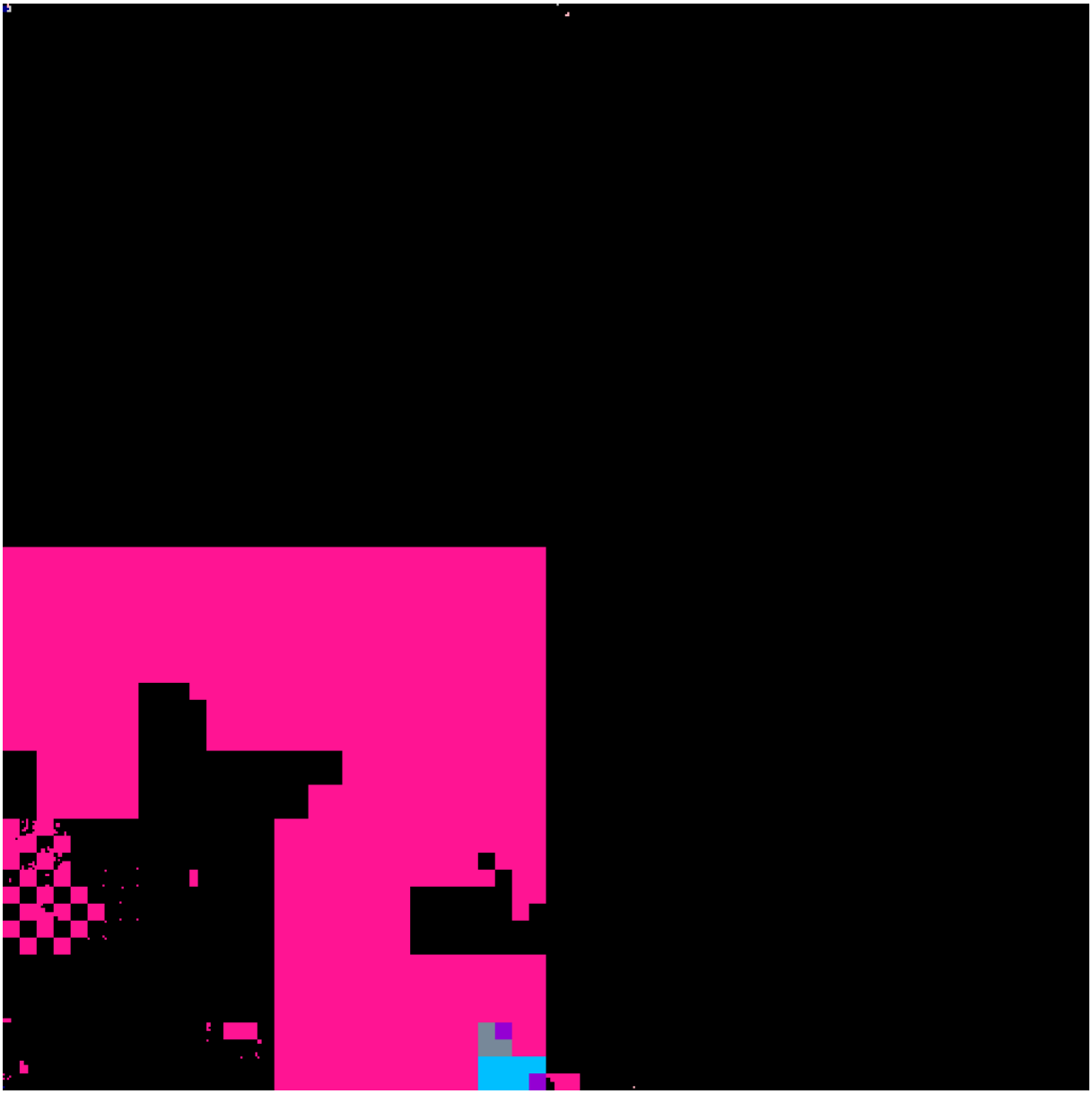
W^X MSM kernel .text with two remaining writable and executable sections on Nexus 4 on Android 4.2.2

Even for newer kernels, such as the 3.4.x MSM kernel on the Nexus 4 and Galaxy S4 examined, there are two read-write executable sections (1 megabyte super-pages). Namely, the start of physical memory at virtual address `0xc000000` is mapped both read-writable and executable. On these newer kernels, it is not possible to directly manipulate the .text section and thereby kernel code executed during normal operation. However, it is trivial to write native code directly to the

executable pages mentioned above. If control flow can later be diverted to an arbitrary kernel address, this allows code execution, even if a specific process' user-space range was not mapped (e.g. when exploiting filesystem flaws using malicious SD cards).

User-land ASLR Bypass

The second interesting observation is more subtle. Upon close inspection, all processes contain a read-only but executable mapping that is accessible from user-land, yet located above the virtual kernel start `0xc0000000`. The page in question is located at `0xffff0000` and it is the ARM interrupt vector page. The ARMv7-A and earlier architectures only support two fixed locations for the interrupt vectors, at virtual addresses `0` or `0xffff0000`. However, the interrupt vector page need not be readable for user-land and for example on iOS on ARMv7-A is not mapped in a way. Because the actual interrupt vectors only occupy the first 64 bytes of the page, the Linux kernel abuses for also placing some user-land helper gadgets in this page. This allows for example accessing the current base address of the Thread Local Storage by simply jumping to a helper at a fixed address within the `0xffff0000` page; that helper gadget then returns this address in `r0`.



init on Nexus 4 with Android 4.2.2 with user executable vectors in the very lower left of the image

Interestingly, the existence of this page is not provided in `/proc/$pid/maps` on older kernels and its accessibility can only be found by parsing the page tables (or reading various ARM kernel or libc sources).

Unfortunately, this executable code at a fixed location can be abused in remote exploitation to bypass ASLR under specific circumstances without requiring an info-leak. There are is one gadget that stands out in the context of exploitation, but there is more code that can be useful depending on the specific target kernel and exploit scenario.

One of the most useful gadgets is the following gem, which is implemented in assembly in the kernel (and therefore stable across compiler versions and settings) and at the fixed offset `+fc0` into the vectors page:

```
fa0:       f57ff05f        dmb     sy
fa4:       e12fff1e        bx      lr
fa8:       e320f000        nop     {0}
...
fc0:       f57ff05f        dmb     sy
fc4:       e1923f9f        ldrex   r3, [r2]
fc8:       e0533000        subs    r3, r3, r0
fcc:       01823f91        strexeq r3, r1, [r2]
fd0:       03330001        teqeq   r3, #1  ; 0x1
fd4:       0afffffa        beq     0xfc4
fd8:       e2730000        rsbs    r0, r3, #0      ; 0x0
fdc:       eafffffef       b       0xfa0
```

This code implements an atomic compare-and-exchange operation for one word. In pseudo-code, it performs the following operation atomically:

```
iff *r2 == r0: *r2 := r1
```

Since it returns normally through the link-register, this gadget can be perfectly used when overwriting a function pointer to a function expecting three parameters. This is especially useful, when another program location needs to be overwritten with a specific value only if the existing value has a certain value.

Consider a scenario, where a function pointer can be overwritten with the address of this gadget and `r0` is user controlled while `r1` and `r2` are program defined pointers. By repeatedly invoking this gadget with different values for `r0` and observing if the pointer pointed to by `r2` is replaced with `r1`, ASLR can be brute-forced safely.

Another simple use of this gadget is writing arbitrary values by directly jumping to `+fcc`, which will store the content of `r1` at the memory location `r2` unconditionally, ignoring `r0` and returning gracefully. This represents a very useful gadget for many C++ Use-after-Free scenarios, where a C++ virtual function table pointer can be hijacked but an arbitrary write to arbitrary memory locations is not directly available.

Local Kernel Information Leak

Since the vectors page contains code to handle software interrupts, it must contain the location of the software interrupt handler. Since this page is user-land readable, the address of the system call handling function is effectively leaked to user-land. The ARM architecture defines the third entry in the vectors page to be a branch to the software interrupt handler and this is indeed the case for the Linux vectors page:

```
8:      e59ff430        ldr     pc, [pc, #1072] ; 0x440
```

Therefore, the address `0xffff0440` contains the kernel virtual address of the system call handler. This address is `0xffff0420` on older kernels, but this can be determined reliably by disassembling the instruction at `0xffff0008`.

In conjunction with the writable kernel `.text` section introduced earlier, this makes local kernel exploitation trivial. An attacker can simply determine the address of the system call handler and introduce a temporary branch instruction to his own code, triggering execution by a software interrupt. Interestingly, the vector page is read-only for the kernel on all examined versions and cannot directly be tampered with.