Droid Rage

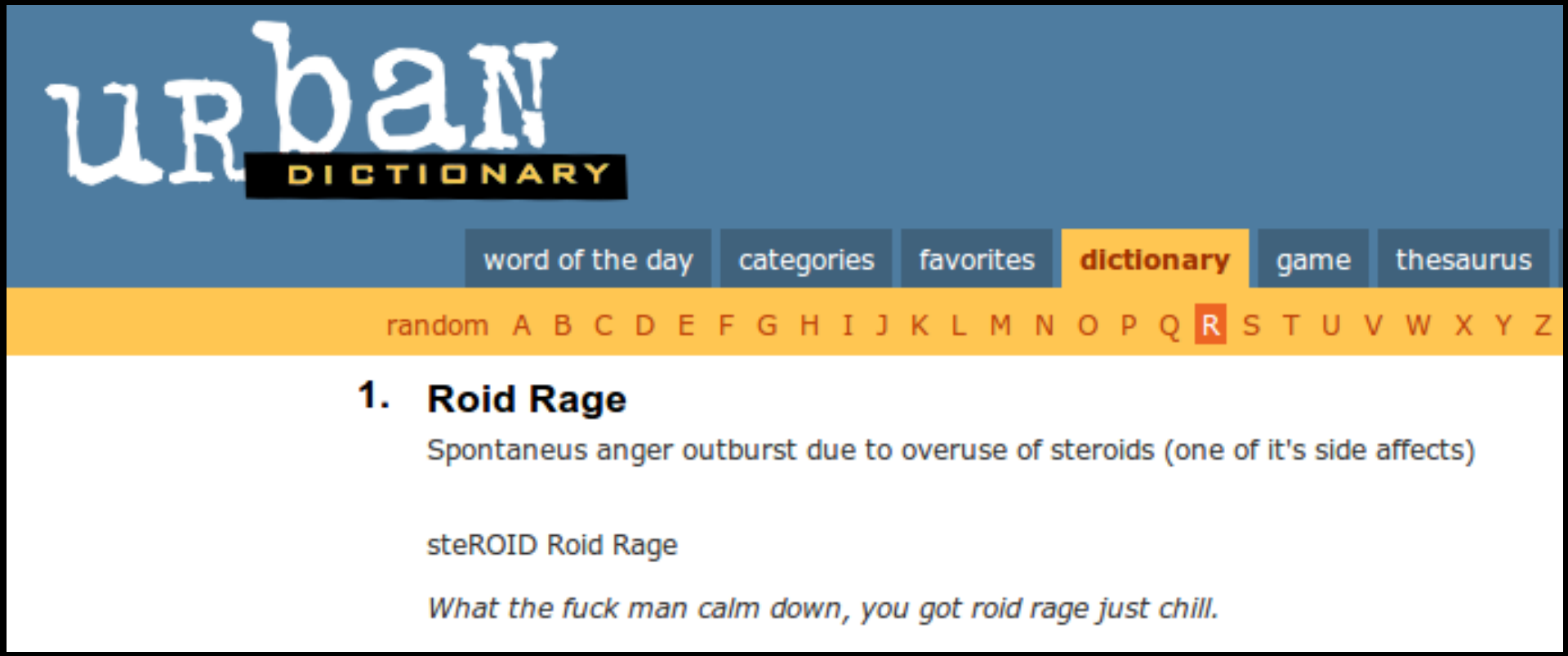
Android exploitation on steroids

Amazing logo by:
http://www.redbubble.com/people/dbenton25

About us

- **Pablo Sole**
  - Twitter: @_pablosole_
  - CTO @ www.remotium.com
- **Agustin Gianni**
  - Twitter - @agustingianni
  - Researcher @ www.remotium.com
- **Remotium** is an innovative and disruptive mobile security technology that secures corporate assets for the bring-your-own-device (BYOD) trend.

What?


urban DICTIONARY
word of the day
categories
favorites
dictionary
game
thesaurus
random A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1. Roid Rage
Spontaneus anger outburst due to overuse of steroids (one of it's side affects)
steROID Roid Rage
What the fuck man calm down, you got roid rage just chill.

Phases of research

- Every new research endeavor starts with the same **three phases**.
- None of which can be avoided.

Phase 1 – Marker Sniffing

Phase 2 - Rage


Breaking News
LIVE
Roid Rage Claims Yet Another Victim.
TVN
EXCLUSIVE

Phase three – feelsgoodman.jpg

Philosophical conclusion

About this talk

- Android browser exploitation guide.
  - Android on **ARM**.
- Debugging
- Hooking
- “Advanced” payload development.
- 100% **bullshit free**.

What is AOSP

- Android Open Source Project
- Gigantic source code base
- Lots of 'interesting' additions:
  - **Google → OK code.**
  - **Samsung → Hilarious code.**
- Building the AOSP is interesting for debugging reasons
  - We will need symbols to correctly debug software

Android SDK & NDK

- **SDK** not necessary for 'binary' analysis but handy for 'managed' tasks.
- **NDK** necessary in order to get working versions of **gdb** and **gdbserver**.
- The **NDK** will get you a nice **tool-chain** for **compilation** and other niceties.
- Installation is easy and it is detailed here:
  - http://developer.android.com/sdk/index.html

The Emulator

- Android comes with an emulator.
- Based on **QEMU**.
- Useful to get to know the system.
- Comes with the **SDK**.
- It can be **built from** the AOSP **sources**.

Warning


Here be dragons!
Circulus Arcticus
TERRA
SAGUENA
Terra
Norombegæ pars
OCEANUS
OCEANUS

Emulator caveats

- **ASLR** and other security features are **turned off**.
  - **Behavior** between the emulator and the physical device will be **different**.
- The emulator is **friendly with undefined instructions**
  - This means that if you find a **ROP** gadget that works on the emulator, you have to make sure it is indeed a well defined instruction.
- Memory **allocations** are not **randomized**.
  - Heap spray, etc. will likely behave very differently on the phone.
- Other than that, the **emulator** is very **useful**.

A message from GDB

Debugging

- Debugging on Android is a **pain in the ass**
- It is hard to get it right the first time
- **Regular GDB does not work**
- A working GDB comes with the NDK
  - This is sub-optimal since that version is a bit dated and **lacks** support for **python scripting**.
  - There may be a newer version in the NDK with python
- ARM **architecture** makes **debugging** very **unnatural**.

ARM & Debugging

- Main execution modes:
  - **ARM**
    - **4 byte** instructions
    - Aligned on a four-byte boundary
  - **THUMB**
    - **2 or 4 bytes** instructions
    - Aligned on a two-byte boundary

ARM Caveats

- Each **execution mode** has a **different** software **breakpoint** instruction.
- The debugger must know which one to use.
- If no symbol information is present, the debugger cannot decide which one to use.
- In general most libraries I've seen are compiled as THUMB.

Exploitation

Browser exploitation

- Android comes with a built in browser based on **WebKit**.
- The **version** of the build **varies** quite a lot between different phones.
- **Rarely up to date.**
  - This means that **1day** bugs killed by the Chrome team can be used for more than 6 months in general.
- The browser is **not sandboxed.**

Browser exploitation

- And there is **Chrome** for Android.
- It comes with a **up to date** version of WebKit.
  - This significantly increases the cost of owning a phone.
- Does come with a **sandbox** implementation.
- Installed by default on some phones.
- It is a fast paced target, lots of changes on each release.
- Generally harder to exploit.

Heaps of problems

Heaps!

- Both the Android browser and Chrome use **known heaps**.
- Android browser uses **dlmalloc**.
- Chrome browser uses **tcmalloc**.
- Both heaps have been widely researched:
  - Attacking the WebKit heap, by Sean Heelan and Agustin Gianni http://immunityinc.com/infiltrate/archives/webkit_heap.pdf
  - Vudo - An object superstitiously believed to embody magical powers, by MaXX Kaempf http://www.phrack.org/issues.html?issue=57&id=8

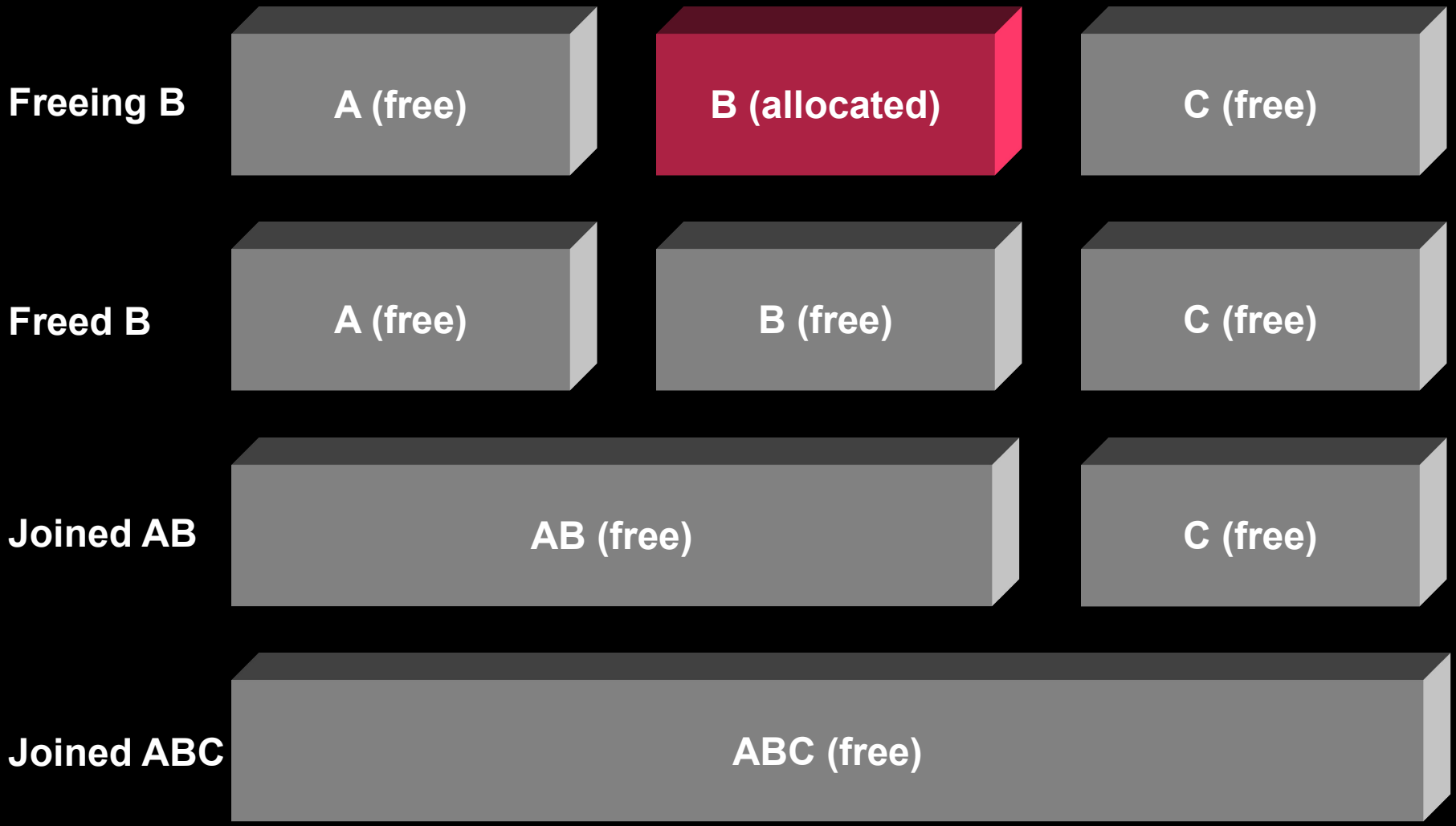
Heaps!

- When working with **dlmalloc** we have to take care of block coalescing.
- Other than that you are golden.
- With **tcmalloc** one can achieve a very good heap state with minimal effort.
- The heap can be 'massaged' by using JavaScript **typed arrays.**
- **Garbage collector** shenanigans are different across both browsers.

Likely heap issues

- **Use-after-free** vulnerabilities are popular these days.
- There is one particular **issue** that needs to be addressed when exploiting them.
- And that is **Block Coalescing.**
- And it **only applies to dlmalloc heap.**

What is coalescing?

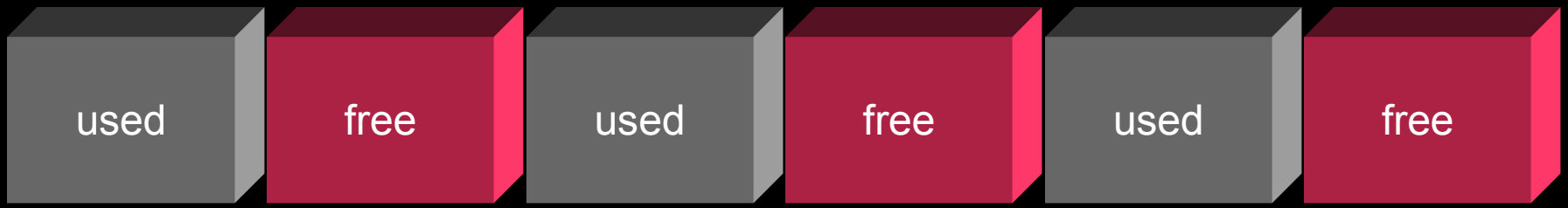


Coalescing Issues

- If two **adjacent blocks** are **free** they will be eventually **coalesced**.
- In a **UAF** situation this will end up modifying the size of the chunk we need to fill.
  - This **can be bad** – The object wont be replaced
  - This **can be good** – We can use another object, bigger with different information.
- We need to control chunk coalescing
  - That is, leaving the heap in a predictable state.

Avoiding coalescing

- Being done for ages now.
- We need to prepare the heap layout in the following way:



Avoiding coalescing

- By making wholes of the size of the chunk we need to fill, we make sure that no coalescing is going to happen.
- Once we decide to free it, it will not be coalesced since it does not have any free neighboring blocks.

Sandboxing

Sandboxing

- Each App has some kind of **sandbox enforced by** the **user ID** of the application.
- Applications cannot mess with other App's files and processes.
- But does not restrict other stuff like opening files, using ioctl, etc.
- Android 4.1 (Jelly Bean) introduces the concept of Isolated Services.

Isolated Services

- A Service is used for doing background processing.
- An Isolated Service runs with its own user ID.
- The isolated process created has no additional groups (ie. no Internet, no SD card access, etc.)
- The Android Service API is used for communication with this process.

Chrome sandbox

- The browser process runs as a **privileged** user.
  - Regular android Activity
- All the renderer's run under a **unprivileged** user.
  - Isolated Services
- This reduces the surface of attack.

Chrome sandbox

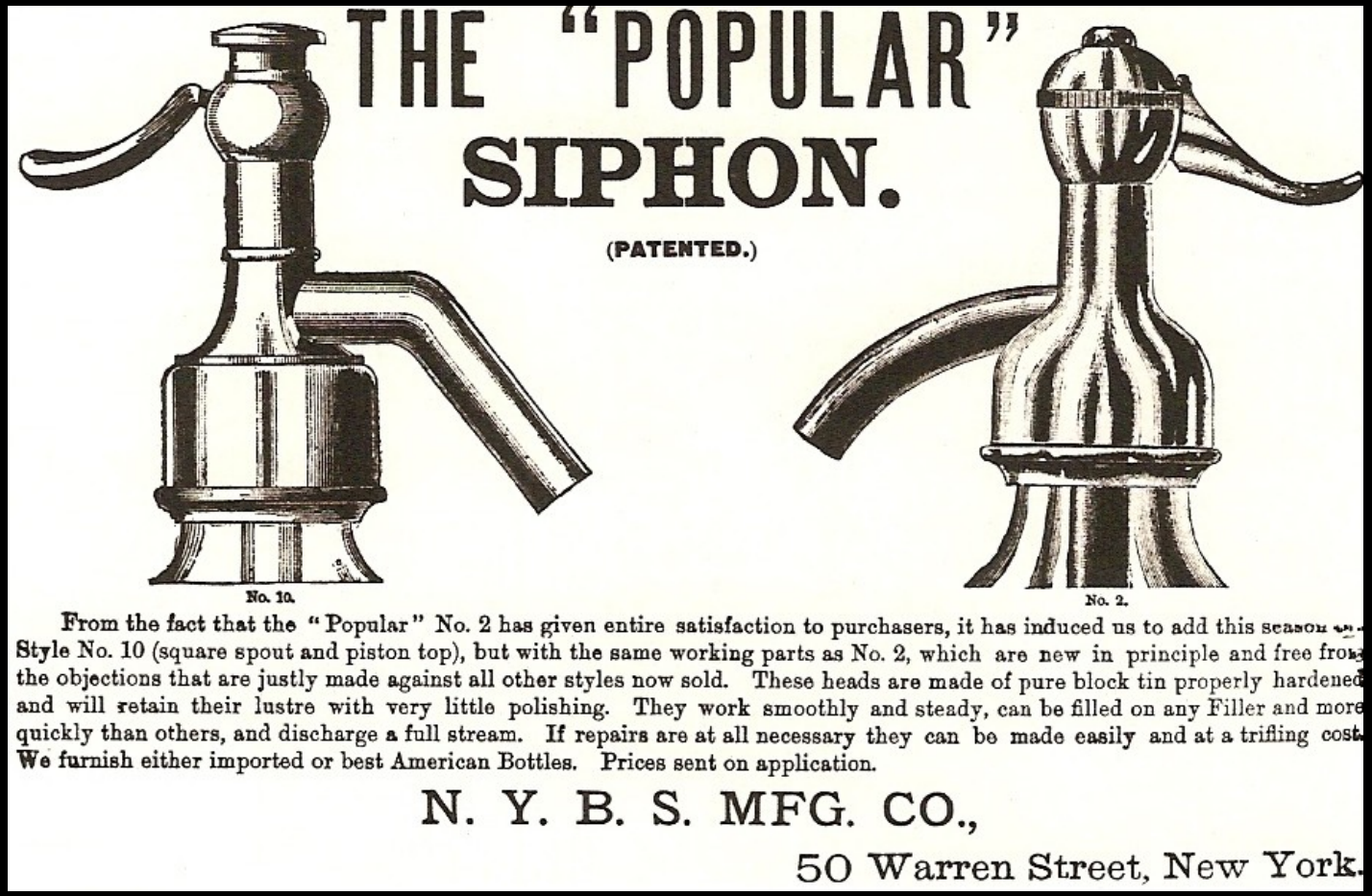
- We **can access the file-system.**
- **Devices** with generous permissions are **accessible** from the sandbox.
- Think about all those **ioctl bugs.**
- So the surface of attack is pretty big:
  - **Kernel** – Linus crap
  - Additional **drivers** – **Vendor crap**
  - Stupid privileged **pipes** / **sockets** used by awful applications.
  - **Rooted devices**!

Sandbox competition!

	Google Chrome	Android Browser	Carbon Rod
Sys-call filters	**NO**	**NO**	**NO**
File-System ACL	**KINDOF**	**NO**	**NO**
Priv. separation	**YES**	**NO**	**NO**
Results	DECENT	DAFUQ	EXPECTED

Results!

Siphoning the sandbox



THE "POPULAR"
SIPHON.
(PATENTED.)

No. 10.

No. 2.

From the fact that the "Popular" No. 2 has given entire satisfaction to purchasers, it has induced us to add this season a Style No. 10 (square spout and piston top), but with the same working parts as No. 2, which are new in principle and free from the objections that are justly made against all other styles now sold. These heads are made of pure block tin properly hardened and will retain their lustre with very little polishing. They work smoothly and steady, can be filled on any Filler and more quickly than others, and discharge a full stream. If repairs are at all necessary they can be made easily and at a trifling cost. We furnish either imported or best American Bottles. Prices sent on application.

N. Y. B. S. MFG. CO.,

50 Warren Street, New York.

Siphoning the sandbox

- In one way or another the sandbox is going to handle privileged information.
- If this information is not properly sanitized, an exploited process can siphon the information out of the renderer process.
- This is a result of renderer processes being a limited resource.
  - On Android there is a hard limit of 6 isolated processes.

Give me the cookies

Sandbox challenges

- A sandbox restricts what we can do inside a process.
- This inherently **increases** the **complexity** of **payloads**.
  - One needs to further **elevate privileges** in order to do useful stuff.
- Coding privilege escalation payloads in assembly is sub-optimal.

Sandbox challenges

- One **could drop a binary** to the exploited system and run it.
  - Generally not possible on the renderer sandbox unless there is a RWX directory on the phone.
- **Dropping binaries** is also a **bad OPSEC** according to TheGrugq.
- One has to do **work** entirely **in memory.**

Payloads on steroids

- The concept is pretty simple:
  - **Load a shared library into the process address space.**
  - **Dynamically link it.**
  - **Execute it.**
- Has been done already on most platforms.
- For more details one can read this:
  - http://grugq.github.io/docs/subversiveld.pdf

Architecture

- Small **C++ binary called the loader.**
  - Load all the information about the process segments.
  - Find libdl.so in memory.
  - Obtain a reference to the linked list of **'struct soinfo'**
  - Resolve useful API like 'dlopen' etc.
- Utility to **blobify the loader** into a shellcode.
  - The loader is a PIC binary.
  - The blobifier will prepare a binary blob based on any binary that when it is loaded straight in memory it can be executed.

Avoiding the assembler

- Some people consider assembly coding 1337.
- I consider writing assembly code a waste of time.
- So we are trying to use c++ to generate our payloads.
- Is this even possible?
- Very much yes!

Avoiding the assembler

- Static code avoids the need of dynamic linking.
- PIC code allows us to load the blob anywhere.
- Avoiding the standard library makes the binary smaller.
- Compiling THUMB code avoids generating code that needs dynamic patching.

```
arm-linux-androideabi-g++          \
    -mthumb                        \
    -fPIC                          \
    -fno-exceptions                \
    -fno-stack-protector           \
    -static                        \
    -fno-rtti                      \
    -fno-ident                     \
    -fconserve-space               \
    -fno-builtin                   \
    -nostdlib                      \
    -Os                            \
    -s                             \
    loader.cpp                     \
    ELFReader.cpp                  \
    libutils.cpp                   \
    syscalls.S                     \
    ctype.cpp                      \
    string.cpp                     \
    vsprintf.cpp                   \
    printk.cpp                     \
    ./libaeabi-cortexm0/uidivmod.S \
    ./libaeabi-cortexm0/uldivmod.S \
    ./libaeabi-cortexm0/idiv.S     \
    ./libaeabi-cortexm0/crt.S      \
    lib1funcs-thumb1.S             \
    -I include                     \
    -fpermissive                   \
    -o loader
```

uLibc

- Since we do not depend on dynamic loading on the loader we need to code our micro-libc.
- This can be easily stolen from bionic.
  - This directory contains the implementation of syscalls: **platform_bionic/libc/arch-arm/syscalls**
- I've implemented open, close, write, mmap and some others.

What's next?

- Once we have a working basic API we need to start to get our building blocks from memory.
- Parsing the /proc/<pid>/ file-system is a reliable way to get information about loaded segments.
  - This assumes that we have permissions to do that.
- By reading the maps entry we can safely read and write memory.
- The next step is getting information about the already linked and loaded binaries.

The 'struct soinfo'

- Linked list of juicy information.
- Has **information** about all the **linked binaries** in memory.
- Allows us to **resolve** any **symbol**.
- And also load any library with a little bit of effort.
- Similar techniques were used in old-school libc exploits.

```
struct soinfo {
public:
    char name[SOINFO_NAME_LEN];
    const Elf32_Phdr* phdr;
    size_t phnum;
    Elf32_Addr entry;
    Elf32_Addr base;
    unsigned size;

    Elf32_Dyn* dynamic;

    soinfo* next;
    unsigned flags;

    const char* strtab;
    Elf32_Sym* symtab;

    Elf32_Rel* plt_rel;
    size_t plt_rel_count;

    Elf32_Rel* rel;
    size_t rel_count;
};
```

Chasing 'struct soinfo'

```
struct soinfo *si = NULL;
const char *needle = "libdl.so";
int nsegments = 0;

nsegments = get_library_segments("linker", segments_, 40);

DEBUG("[i] Found %x segments\n", nsegments);

for (i = 0; i < nsegments; i++) {
    void *beg = (void *) segments_[i].beg;
    void *end = (void *) segments_[i].end;

    size_t size = (size_t) end - (size_t) beg;
    si = (struct soinfo *) memmem((void *) segments_[i].beg, size, needle,
            strlen(needle));

    // Check if we got the right structure.
    if (si != NULL && si->flags == FLAG_LINKED && si->nbucket == 1) {
        return si;
    }
}
```

Resolving symbols

```
static void *find_symbol(const char *library, const char *symbol) {
    struct soinfo *si = find_loaded_library(library);
    if (!si) {
        DEBUG("[e] Could not get soinfo for %s, aborting ...\n", library);
        return NULL;
    }

    int i;
    for (i = 0; i < si->nchain; i++) {
        Elf32_Sym* sym = &si->symtab[i];
        if (!strcmp(si->strtab + sym->st_name, symbol) && sym->st_value) {
            return (void *) sym->st_value;
        }
    }

    return NULL;
}
```

- Get a reference to the appropriate soinfo structure.
- Traverse the symbol table and compare the name of the function.

What's next now?

- So we have the address of any exported symbol.
  - This gives us great flexibility.
- How can we load our shared object into memory?
  - If you said using **dlopen** you are **wrong**.
  - We cannot touch the file-system.
  - dlopen relies on the file-system to load a binary.

What do?

- Turns out it is a little bit more complicated.
- We need to **manually link** and **load** our **shared object**.
- Patching open, read and other functions used by dlopen to emulate the load from memory is a good option.
  - See this paper for more information on this technique: http://hick.org/code/skape/papers/remote-library-injection.pdf
- I've decided to steal^W**reimplement** the linker.

Stealing the wheel

Stealing the wheel

- We do not really need to re-implement the wheel.
- We can patch the wheel.
- Modifying the linker that comes with **bionic** (Androids libc implementation) is the easiest option.
- https://github.com/android/platform_bionic

Stealing the wheel

- **Android linker** is really **self contained**.
- Most of the functionality is located under the 'linker' directory on the bionic sources.
  - Even an ELF parser.
- The single most important thing that we need to modify is the **ElfReader** class within linker_environ.cpp file.

Poking the ELF

- We need to make the ElfReader read from memory instead of a file descriptor.
- There are a few places we need to change:
  - `ElfReader::ReadElfHeader()`
  - `ElfReader::ReadProgramHeader()`
  - `ElfReader::LoadSegments()`

ReadELFHeader

```
bool ElfReader::ReadElfHeader() {
    if (!elf_) {
        ssize_t rc = ::read(fd_, &header_, sizeof(header_));
        if (rc < 0) {
            printk("can't read file \"%s\"", name_);
            return false;
        }

        if (rc != sizeof(header_)) {
            printk("\"%s\" is too small to be an ELF executable", name_);
            return false;
        }
    } else {
        memcpy(&header_, elf_, sizeof(header_));
    }

    return true;
}
```

ReadProgramHeader

```
bool ElfReader::ReadProgramHeader() {
    phdr_num_ = header_.e_phnum;
    Elf32_Addr page_min = PAGE_START(header_.e_phoff);
    Elf32_Addr page_max = PAGE_END(header_.e_phoff + (phdr_num_ * sizeof(Elf32_Phdr)));
    Elf32_Addr page_offset = PAGE_OFFSET(header_.e_phoff);

    phdr_size_ = page_max - page_min;

    void* mmap_result;

    if (elf_) {
        mmap_result = reinterpret_cast<void *>(elf_);
    } else {
        mmap_result = mmap(NULL, phdr_size_, PROT_READ, MAP_PRIVATE, fd_, page_min);

        if (mmap_result == MAP_FAILED ) {
            printk("\"%s\" phdr mmap failed", name_);
            return false;
        }
    }

    phdr_mmap_ = mmap_result;
    phdr_table_ = reinterpret_cast<Elf32_Phdr*>(reinterpret_cast<char*>(mmap_result)
            + page_offset);

    return true;
}
```

LoadSegments

```
void* seg_addr;
if (!elf_) {
    seg_addr = mmap((void *) seg_page_start, file_end - file_page_start,
            PFLAGS_TO_PROT(phdr->p_flags), MAP_FIXED | MAP_PRIVATE, fd_,
            file_page_start);
} else {
    // Here we add the write protection due to the fact that we
    // have to copy the contents from one mapping to the other.
    // Also we make sure .text relocations work.
    seg_addr = mmap((void *) seg_page_start, file_end - file_page_start,
            PFLAGS_TO_PROT(phdr->p_flags) | PROT_WRITE,
            MAP_ANONYMOUS | MAP_FIXED | MAP_PRIVATE, 0, 0);

    memcpy(seg_addr, elf_ + file_page_start, file_end - file_page_start);
}
```

Exploitation demo!

- We will see how we can get a remote root shell on a up to date Chrome Browser running on a Samsung S2 Android phone.

Conclusions

- Android is non homogeneous and moving target.
- There is a ton of work being done on securing the platform.
  - Expect work being done on hardening
    - SELinux
    - grsecurity / pax
    - Chrome will get a better sandbox
      - They work quick so there is a chance that they've already done while I was researching this.
- The cost of owning a phone will raise in the next two years.
- We can expect more 'managed' vulnerabilities.

Finale!

- I fucking hate smart phones now.
- Everybody should buy a brick :P

Questions?

Thanks (alphabetical order)

- Georg Wicherski - @ochsff
- Joshua Drake - @jduck
- Sinan Eren - @remotium
- The grugq - @thegrugq

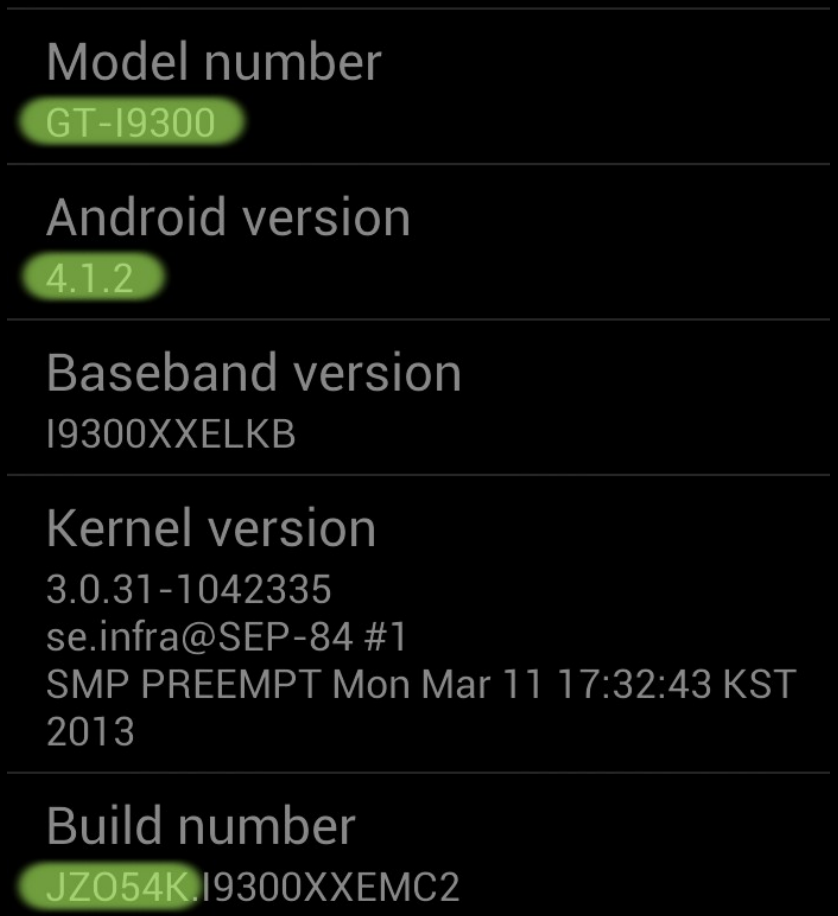
Appendix of not so fun stuff

- Probably incomplete stuff but useful nonetheless

Building AOSP

- Needs a relatively new system
  - Ubuntu >= 10.04 is recommended
  - I'm using 13.04
  - Only weird requirement is **Oracle JDK 6**
    - https://launchpad.net/~webupd8team/+archive/java
  - Additional requirements are listed here:
    - http://source.android.com/source/initializing.html
- Complete instructions can be found here:
  - http://source.android.com/
- There are multiple releases:
  - http://source.android.com/source/build-numbers.html

Getting the right branch



- GT-I9300 is Samsung Galaxy S3.
- 4.1.2 is the Android version.
- JZO54 is my build number.

Getting the right branch

- With the build number one can get the branch name at:
  - http://source.android.com/source/build-numbers.html#source-code-tags-and-builds

JRO03L	android-4.1.1_r4	Nexus S
JRO03O	android-4.1.1_r5	Galaxy Nexus
JRO03R	android-4.1.1_r6	Nexus S 4G
JRO03S	android-4.1.1_r6.1	Nexus 7
JZO54K	android-4.1.2_r1	Nexus S, Galaxy Nexus, Nexus 7
JZO54L	android-4.1.2_r2	
JZO54M	android-4.1.2_r2.1	

Getting the sources

- The most important thing is to get the build name right.
- The 'sync' will take a couple of **hours** to finish.

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ mkdir android_master
$ cd android_master
$ repo init -u https://android.googlesource.com/platform/manifest -b <YOUR_BUILD_NAME>
$ repo sync
```

Building

- There are multiple targets to build
  - The most interesting for bug finding is the debug build.
  - For more information on build targets see:
  - http://source.android.com/source/building-running.html#choose-a-target

```
$ # Initialize the environment with the envsetup.sh script.
$ source build/envsetup.sh
$ # Choose which target to build with lunch.
$ # "full-eng" does a complete build of the emulator with all debugging enabled.
$ lunch full-eng
$ # Compilation with 16 simultaneous processes.
$ make -j16
```

Running the emulator

- The build system will add the emulator to the PATH.
- That mean running 'emulator' will utilize the recently build Android system.

```
$ # Run the emulator
$ emulator
```

Preparing the NDK tool-chain

- Once the SDK and the NDK are installed you need to add them to the $PATH variable.

```
if [ -d "$HOME/android" ] ; then
    PATH="$HOME/android/android-sdk/platform-tools:$HOME/android/android-sdk/tools:$HOME/android/android-ndk:$PATH"
fi

if [ -d "$HOME/android-toolchain" ] ; then
    PATH="$HOME/android-toolchain/bin:$PATH"
fi
```

Preparing the NDK tool-chain

- This will create a full **build system**.
- One can use the tool-chain to **cross-compile** software.
- The tool-chain contains also a link to the correct GDB version.

```
$ ./android-ndk/build/tools/make-standalone-toolchain.sh \
    --install-dir=/home/anon/android-toolchain \
    --system=linux-x86_64
  Auto-config: --toolchain=arm-linux-androideabi-4.6
  Copying prebuilt binaries...
  Copying sysroot headers and libraries...
  Copying libstdc++ headers and libraries...
  Copying files to: /home/anon/android-toolchain
  Cleaning up...
  Done.
```

Android tool-chain

```
anon@research:~$ arm-linux-androideabi-

    arm-linux-androideabi-addr2line  arm-linux-androideabi-g++
    arm-linux-androideabi-ld         arm-linux-androideabi-ranlib
    arm-linux-androideabi-ar         arm-linux-androideabi-gcc
    arm-linux-androideabi-ld.bfd     arm-linux-androideabi-readelf
    arm-linux-androideabi-as         arm-linux-androideabi-gcc-4.6
    arm-linux-androideabi-ld.gold    arm-linux-androideabi-run
    arm-linux-androideabi-c++        arm-linux-androideabi-gcov
    arm-linux-androideabi-ld.mcld    arm-linux-androideabi-size
    arm-linux-androideabi-c++filt    arm-linux-androideabi-gdb
    arm-linux-androideabi-nm         arm-linux-androideabi-strings
    arm-linux-androideabi-cpp        arm-linux-androideabi-gdbtui
    arm-linux-androideabi-objcopy    arm-linux-androideabi-strip
    arm-linux-androideabi-elfedit    arm-linux-androideabi-gprof
    arm-linux-androideabi-objdump
```

Debugging tools

- The Android toolchain comes with gdb
  - arm-linux-androideabi-gdb
- The emulator comes with a gdbserver already installed.
- Generally devices do not have it.
  - The binary can be found on the NDK here:
  - **prebuilt/android-arm/gdbserver/gdbserver**
- Pushing the binary into the phone:
  - $ adb push /data/local/tmp

Debugging tools

- GDB is useful for source level debugging.
- But it lacks a lot of properties useful for binary debugging.
- Enter IDA Pro.
- IDA Pro comes with a debug server that works pretty nicely on Android.
- You can find it on the the IDA installation directory under the name **'android_server'**

ADB Interface

- Short for Android Debug Bridge
- Single most useful tool for working with Android
- **Pull** and **push** files from the device/emulator.
- **Forward ports** between the emulator/device and our local system.

ADB - Commands

- $ adb push <file> <remote_dir>
- $ adb pull <file>
- $ adb forward <port>:tcp <port>:tcp
- $ adb shell
- $ adb install <file.apk>
- $ adb uninstall <package.name>