Bootstrapping Kindle research for the lazy attacker

Tanguy Dubroca

LeHack 2026

27/06/2026

whoami

- **Tanguy "Sideway" Dubroca**
  - Vulnerability Researcher @ **Synacktiv**
  - Interested in kernel exploitation
  - Avid book reader, and long time kindle owner !
- **Synacktiv**
  - Offensive security company
  - ~200 ninjas
  - We are hiring !

Ordre du jour

- **Kindle jailbreak on recent firmwares**
  - 5.18.6 at the time
- **Less about the bugs and exploits**
- **More about the mindset and the thought process**

Prologue

Why this talk ?

- **Console hacking (3DS) got me interested in VR**
  - Found the concept of executing unsigned code on a closed device really cool
  - I understood the theory and technique behind exploits
  - But I did not understand how they bootstrapped their research
- **Bootstrapping problem**
  - The code on the 3DS is encrypted
  - You need the code or some introspection to develop an exploit …
  - … but you need an exploit to get the code or introspection
- **The project**
  - Wanted to tackle the bootstrapping problem myself on a closed down device

Target selection: Constraints

- **I am already doing VR full time at work**
  - I don't want to spend all of my free time on a VR project
  - But I also really want to solve the bootstrapping problem
  - I needed to find a target hitting the difficulty sweet spot
- **Wishlist**
  - The target must not be too hard (ex: mobile phone)
  - The target must not be too soft (ex: random IoT device)
  - The target must not be too costly
  - The target must not have obfuscation
  - The target must not require hw hacks

Target selection: Constraints

Kindle

- **Satisfies all of my constraints**
  - Not too costly for standard models (100~200€)
  - Target on the softer side, as it is regularly jailbroken
  - But not too soft, as there are periods without jailbreaks
  - Firmware is freely available on Amazon's website
  - Open source tools exist to unpack the firmware (**KindleTool**)

Kindle jailbreak project

- **Goal**
  - Write a kindle jailbreak on latest firmware
  - Write the jailbreak without relying on an existing jailbreak for debugging
- **Research target**
  - Kindle Paperwhite 5 (2021) on firmware **5.18.4** / **5.18.6**

Chapter 1: Attack surface selection

Kindle firmware

- **OS**
  - Classic embedded linux userland
  - Two users: framework (uid 9000) and root (uid 0)
  - Interface in React Native
  - Some apps / menus are web pages
  - System components implemented in both java and native code
  - Everything tied through an ipc bus: **lipc**

Kindle firmware

Attack surface selection

- **Poor man's strategy**
  - Look at existing exploits and note their entrypoint
- **Public jailbreaks**
  - **Nosebleed**: Chrome exploit (javascript)
  - **AdBreak**: WebKit exploit (javascript)
  - **WinterBreak**: logic bug / feature
  - **KindleBreak**: WebKit exploit (media parsing)
  - **LanguageBreak**: command injection
- **Public research**
  - https://research.checkpoint.com/2021/i-can-take-over-your-kindle/: reader (media parsing)
  - https://blog.thalium.re/posts/dont-judge-an-audiobook-by-its-cover-taking-over-your-amazon-account-with-a-kindle/: scanner (media parsing)

Attack surface selection: Browsers

- **The kindle bundles (very) old browsers**
  - **Chrome**: Build from 2019
  - **WebKit**: Build from 2011~2012 !
  - Huge probability that some 1-days could be used !
- **JS exploits are comfortable**
  - You can execute your own code
  - Bugs give nice, scriptable primitives (ex: arbitrary r/w)
  - Compare this to file format exploits …
- **Experimentation is easy**
  - A lot of resources about browser exploitation are available (blogposts, talks)
  - Old browser builds can debugged locally
  - The only hard work is porting everything blind once it works

Attack surface selection

- **Browser exploitation looks comfortable and popular**
  - I picked Chrome as an entrypoint for the jailbreak

Chapter 2: Chrome RCE

Information gathering

- **We need a few pieces of information**
  - What is the version of chrome on the kindle ?
  - Which runtime options are enabled / disabled ?
  - Which chrome bugs could work on the target version and constraints ?
  - How do we find old builds for local testing ?
  - How do we test on the kindle ?

Chrome version

- **Getting the binaries**
  - Chrome bundled as squashfs in `/usr/bin/chromium.sqsh`
  - With strings and double checking with binja the exact version is `Chrome 75.0.3770.143`

Runtime options

- **Execution**
  - Chrome is launched through the `/usr/bin/browser` wrapper script

```
#!/bin/bash

...

exec chroot /chroot /usr/bin/chromium/bin/kindle_browser \
     --no-zygote --no-sandbox --single-process \
     --skia-resource-cache-limit-mb=64 \
     --disable-gpu --in-process-gpu --disable-gpu-sandbox --disable-gpu-compositing \
     --enable-dom-distiller --enable-distillability-service --force-device-scale-factor=2 \
     --js-flags="jitless" \
     [...] \
     --user-agent="Mozilla/5.0 (X11; U; Linux armv7l like Android; en-us) AppleWebKit/531.2+[...]" \
```

Runtime options

- **The good**
  - Chrome sandbox disabled through `--no-sandbox`
  - JavaScript RCE == RCE as a 'privileged' process
- **The bad**
  - `--js-flags=jitless` disables the JIT pipeline in v8
  - Most public bugs, pocs, and exploits target the JIT …
  - Even without sandbox we are still running in a chroot

Looking for bugs

- **Requirements**
  - Pure JS engine bugs in priority
  - DOM and third party bugs if we don't have a choice
  - JIT bugs are banned (as it is disabled)
- **Three strategies**
  1. Manual grepping through bug reports
  2. Ask an LLM
  3. Ask around for help

Strategy #1: Manual work

- **Plan**
  - Look for all CVE's affecting Chrome 75~80
  - Sort them by surface (JS > DOM > 3rd party)
  - Prioritise CVE's with public PoCs (ex: Project Zero itw pocs, public crbug issues)
  - Rinse and repeat ...
- **Results**
  - No JS bugs
  - 2 DOM+third party bugs

Strategy #2: LLM

- **Plan**
  - Asked ChatGPT deep research to look around
- **Results**
  - Found pretty much the same things I discovered manually …
  - … and some bugs I had missed !
  - Discovered some crashing PoCs but nothing exploitable

Strategy #3: Please help

- **Plan**
  - Some of my colleagues really like JS engines
  - Asked them to become my pro-bono exploit brokers
  - Gave me pretty cool tips and bugs I had missed !
  - Gave me PoCs to try to out
- **Results**
  - One the PoCs worked (**CVE-2022-1364**)

CVE-2022-1364

- **Colleague**: sends me the PoC
- **Me**: tries out the poc
- **Me**: it works
- **Me**: "nice thanks"
- **Colleague**: "wait… this is a JIT bug !"
- **Me**: ???????

JIT ?

```
...
exec chroot /chroot /usr/bin/chromium/bin/kindle_browser \
     --no-zygote --no-sandbox --single-process \
     --skia-resource-cache-limit-mb=64 \
     --disable-gpu --in-process-gpu --disable-gpu-sandbox --disable-gpu-compositing \
     --enable-dom-distiller --enable-distillability-service --force-device-scale-factor=2 \
     --js-flags="jitless" \
```

- **Not so jitless**
  - JIT supposed to be disabled with `--js-flags="jitless"`
  - But the syntax is wrong, it should be `--js-flags="--jitless"` instead
  - V8 ignores the option and leaves the JIT enabled !

CVE-2022-1364

- **Exploit**
  - PoC link: https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2022/CVE-2022-1364.html
  - JIT optimization bug in turbofan
  - PoC leaks a special **TheHole** value to javascript
  - Well known primitive so generic exploit techniques exist
  - Decided to write the exploit myself for learning !

CVE-2022-1364: Exploit

- **Used a known strategy based on Set/Map with holes**
  - Deleting entries using the hole as index has side effects
  - Leads to an OOB write from the Set/Map backing store
  - We can shoot the size of an array and get an OOB R/W
- **High-level exploit recipe**
  - Allocate the map
  - Allocate arrays after the map's backing store
  - Leak the hole
  - Use the OOB write to shoot one of the array's length
  - Build fakeobj, addrof, arbitrary R/W
  - Shoot a wasm module JIT function with shellcode using the R/W
  - ...
  - Profit ?

Exploit development process

- **Looked for a desktop chrome of the same version**
  - Tried to build an old V8
  - Quickly abandonned this idea …
  - Downloaded a close enough build from chromium build archive
    - https://commondatastorage.googleapis.com/chromium-browser-snapshots/index.html?prefix=Linux_x64/652428/
- **Developped the first part on desktop**
  - Used V8 native (`%DebugPrint`) to print object adresses
  - Used gdb to explore data structures in memory
  - Followed through the V8 source code
  - **VERY** helpful to have introspection to get comfortable with the implementation
  - Stopped once I had a corrupted array

Exploit development process

- **Kindle network setup**
  - Fake AP + webserver on a laptop
  - Firewall rules allowing only outbound DNS + connectivity check
  - Needed as Kindles automatically update themselves **:)**
- **Ported the PoC to the kindle**
  - Had to do some guessing as a few offsets were different
  - Wrote the rest of the exploit completely on the kindle
  - Had to tediously debug the whole thing through websocket logging:')
  - In the end we get (unreliable) shellcode execution !

Exploit development process

Exploit development process

- **Issues with the initial exploit setup**
  - Code exec is unreliable + a manual multi-step process
  - Raw machine code is not practical for development
  - We cannot iterate quickly

Exploit development process

- **Comfy setup: ushell**
  - Initial shellcode implemented in C
  - Connects back to a laptop
    - Port 4000 to stream stdout
    - Port 4001 to receive commands
  - Implements a persistent in-memory ELF loader over the network
    - Fork + "Exec" the loaded binaries
    - Some surprises with relocations ...
  - Implements debug and release profiles

Exploit development process

- **Advantages**
  - Developing in a high-level language like C is nice
  - The setup allows for quick iteration
  - The code can be debugged locally with `qemu-user-arm`

Chapter 3: LPE

Attack surface exploration

- **Current situation**
  - We have stable code exec in the browser process **:)**
  - We are running as uid 9000 (framework) and gid 0 (root) **:o**
  - We are inside a chroot **:(**
  - What can we reach ?

Attack surface exploration

- **Chroot configuration**
  - Configured at system startup
  - Implemented as helper function in `/etc/upstart/chroot.conf`

```
mkdir -p /tmp/chroot || true
mkdir -p /chroot/dev || true
mkdir -p /chroot/var/local || true
mkdir -p /chroot/mnt/us || true

if [ $(lipc-get-prop com.lab126.volumd driveModeState) == 0 ]; then
    chroot_mount
fi

mount -o bind /dev /chroot/dev || true
```

```
 chroot_mount()
{
    # ...
    mount -o bind /var/local /chroot/var/local || true
    mount -o bind /mnt/us /chroot/mnt/us || true
    # ...
}
```

Attack surface exploration

- **Potential attack surface**
  - **Kernel** through the `/dev` devices + syscalls
  - **System services** through folders mounted as rw
  - **System services** through lipc

Attack surface exploration

- **Kernel**
  - I like kernel exploitation so I went there first
  - Some drivers are accessible as we are gid 0
  - `/proc/kmsg` accessible as we are gid 0
  - Did some preliminary reverse-engineering
  - Found some concerning crashes (null deref on device open…)
  - Could be an interesting attack surface for later

Attack surface exploration

- **System services through folders**
  - Listed all files writable from our context
  - `/var/local` folder is accessible
  - Contains system services configuration files and databases
  - Revived an old LPE as a chroot escape + LPE !
- **System services through lipc**
  - Did not have time to explore the surface
  - Used by all recent jailbreaks for LPE
  - Could be an interesting attack surface for later

LPE: appmgrd

- **appmgrd**
  - System service responsible for launching system applications
  - Application launch configuration stored in `/var/local/appreg.db`
  - Application launch can be triggered through lipc
  - Runs as root

LPE: appmgrd

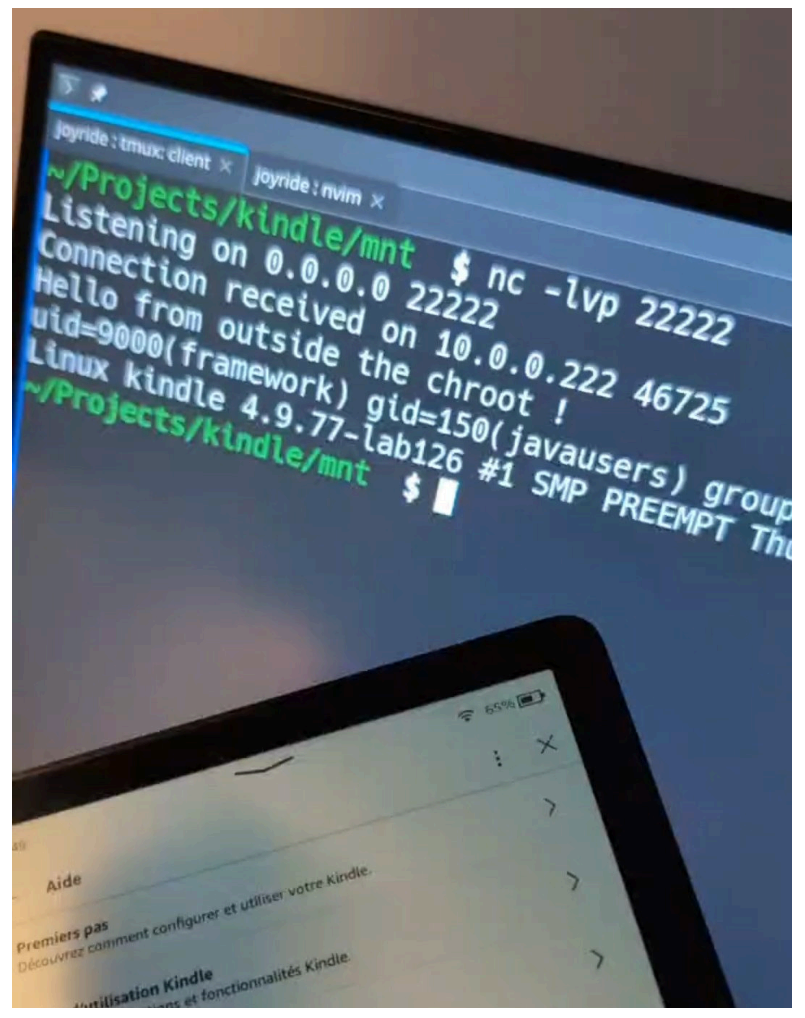
	handlerId	name	value
	Filter	Filter	Filter
1535	dcd	cmd.get.newDeviceCredentials.params	deviceSerialNumber,radioId,secret,secondaryRadioId,reason,deviceRequestVerificationD…
1536	dcd	cmd.registerDevice.optional.params	
1537	dcd	cmd.registerDeviceWithToken.optional…	
1538	dcd	cmd.get.newDeviceCredentials.optiona…	
1539	dcd	btfd.btpowersave.timeout	1200
1540	com.lab126.subscription	command	/usr/bin/mesquite -l com.lab126.subscription -c file:///var/local/mesquite/…
1541	com.lab126.subscription	unloadPolicy	unloadOnPause
1542	com.lab126.subscription	framework	
1543	com.lab126.browser	command	/usr/bin/browser -j
1544	com.lab126.browser	framework	
1545	com.lab126.browser	unloadPolicy	unloadWhenNeeded
1546	com.lab126.browser	maxExecTime	30
1547	com.lab126.browser	maxLoadTime	30
1548	com.lab126.browser	maxUnloadTime	10
1549	com.lab126.csapp	command	/usr/bin/mesquite -l com.lab126.csapp -c file:///var/local/mesquite/csapp/

LPE: appmgrd

- **Attack plan**
  - Change the command line of an application to a reverse shell
  - Launch the backdoored application through lipc
  - …
  - Profit ?
- **Old technique**
  - Already used in 2021 by CheckPoint research
    - https://research.checkpoint.com/2021/i-can-take-over-your-kindle/

LPE: appmgrd

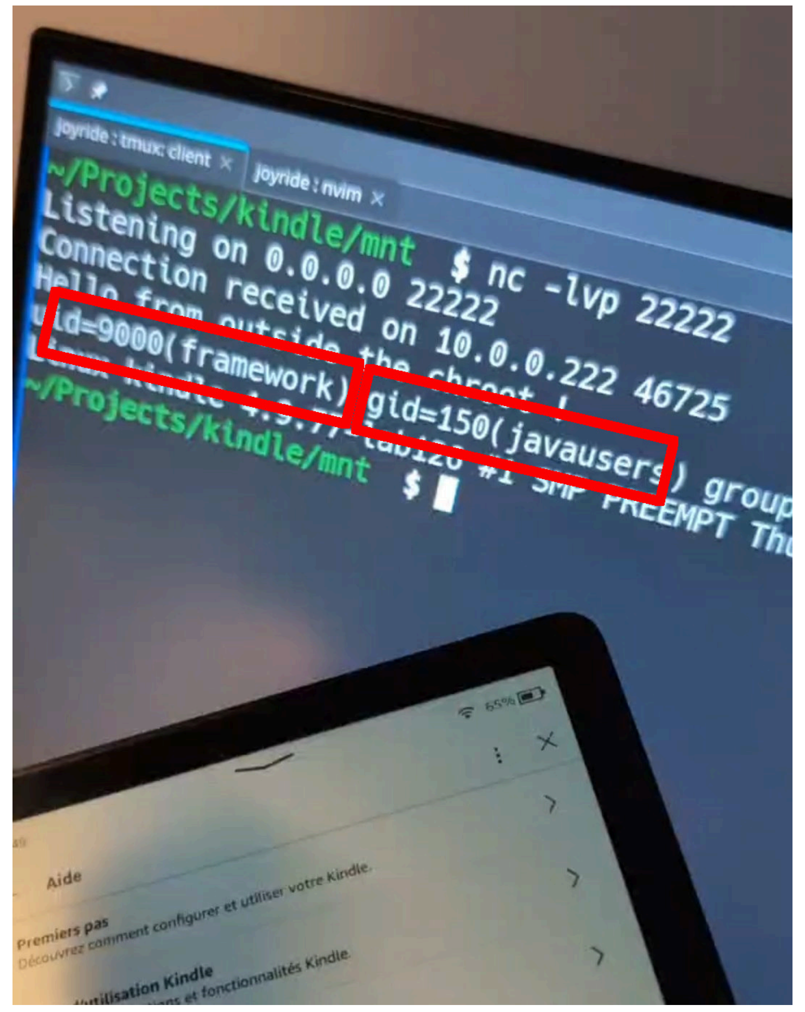
- **Root !!!**



LPE: appmgrd

SYNACKTIV

- Root ???



LPE: appmgrd

- **Not root ?**
  - We get a shell outside the chroot
  - But we only run as framework
  - Why ?

LPE: appmgrd

- **libappmgrd**
  - Library implementing most of `appmgrd` logic
  - Implements privilege dropping to framework…

LPE: appmgrd

```c
int32_t *appmgr_ensure_running(char *lipc_app_id, char *executable)
{
    /* ... */
    const char *path = argv[0];

    for (int i = 0; i < 0x100 && argv[i] != NULL; i++) {
        if (g_str_equal(argv[i], "-j") || g_str_equal(argv[i], "--jail")) {
            has_jail_arg = 1;
            break;
        }
    }

    is_special_executable = strcmp(path, "/usr/bin/mesquite") == 0 || strcmp(path, "/usr/bin/browser") == 0;

    if (!has_jail_arg || !is_special_executable) {
        syslog(LOG_NOTICE, "I def:appmgr_proc:Setting Framework userid:");
        pw = getpwnam("framework");

        if (pw != NULL) { user = pw->pw_name; uid = pw->pw_uid; gid = pw->pw_gid; }

        if (setgid(gid) != 0 || setuid(uid) != 0) {
            /* ... */
        }
    }

    execve(argv[0], argv, envp);
}
```

LPE: appmgrd

```c
int32_t *appmgr_ensure_running(char *lipc_app_id, char *executable)
{
    /* ... */
    const char *path = argv[0];

    for (int i = 0; i < 0x100 && argv[i] != NULL; i++) {
        if (g_str_equal(argv[i], "-j") || g_str_equal(argv[i], "--jail")) {
            has_jail_arg = 1;
            break;
        }
    }

    is_special_executable = strcmp(path, "/usr/bin/mesquite") == 0 || strcmp(path, "/usr/bin/browser") == 0;

    if (!has_jail_arg || !is_special_executable) {
        syslog(LOG_NOTICE, "I def:appmgr_proc:Setting Framework userid:");
        pw = getpwnam("framework");

        if (pw != NULL) { user = pw->pw_name; uid = pw->pw_uid; gid = pw->pw_gid; }

        if (setgid(gid) != 0 || setuid(uid) != 0) {
            /* ... */
        }
    }

    execve(argv[0], argv, envp);
}
```

LPE: appmgrd

- **But there are exceptions !**
  - Browsers are executed as root when launched with the `-j` / `--jail` flag
  - They need root to implement their own sandboxing (ex: chroot)
  - Could we make **browser** or **mesquite** load custom code ?

LPE: appmgrd

```
mesquite --help-all
Usage:
  mesquite [OPTION...] - Mesquite Daemon

Help Options:
  -h, --help                           Show help options
  --help-all                           Show all help options
  --help-gtk                           Show GTK+ Options

GTK+ Options
  --class=CLASS                        Program class as used by the window manager
  --name=NAME                          Program name as used by the window manager
  --screen=SCREEN                      X screen to use
  --sync                               Make X calls synchronous
  --gtk-module=MODULES                 Load additional GTK+ modules
  --g-fatal-warnings                   Make all warnings fatal

Application Options:
  -d, --daemonId=lipcId                lipc Id for this instance of wafapp
  -l, --lipcId=lipcId                  lipc Id for this instance of wafapp
  -c, --contentLocation=location       location of the application content
  -u, --userId=userId                  user id to be set
  -j, --jail=jail                      set it to run in chroot jail
  --display=DISPLAY                    X display to use
```

LPE: appmgrd

```
mesquite --help-all
Usage:
  mesquite [OPTION...] - Mesquite Daemon

Help Options:
  -h, --help                         Show help options
  --help-all                         Show all help options
  --help-gtk                         Show GTK+ Options

GTK+ Options
  --class=CLASS                      Program class as used by the window manager
  --name=NAME                        Program name as used by the window manager
  --screen=SCREEN                    X screen to use
  --sync                             Make X calls synchronous
  --gtk-module=MODULES               Load additional GTK+ modules
  --g-fatal-warnings                 Make all warnings fatal

Application Options:
  -d, --daemonId=lipcId              lipc Id for this instance of wafapp
  -l, --lipcId=lipcId                lipc Id for this instance of wafapp
  -c, --contentLocation=location     location of the application content
  -u, --userId=userId                user id to be set
  -j, --jail=jail                    set it to run in chroot jail
  --display=DISPLAY                  X display to use
```

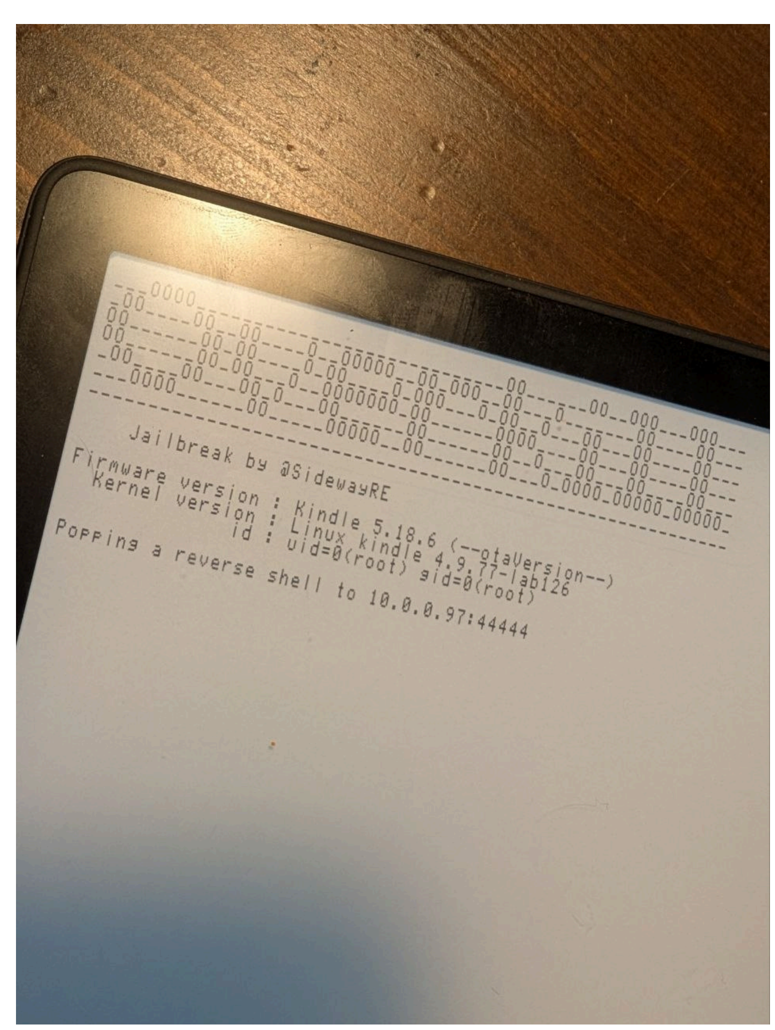
LPE: appmgrd

- **Mesquite**
  - Uses gtk, which loads the gtk runtime libraries
  - The runtime hooks into arguments parsing to inject its own options
  - `--gtk-module` loads a "module", which is nothing more than a shared library !

Chroot escape + LPE

- **Step 1: Patch `appreg.db` with chroot escape command**
  - Command copies stage2 from `/mnt/us` (noexec) to `/tmp` (exec)
  - Trigger exec through lipc
- **Step 2: Patch `appreg.db` with LPE command**
  - `/usr/bin/mesquite --gtk-module=/tmp/payload.so -j`
  - Trigger exec a second time through lipc
- **…**
- **Win !**

LPE: appmgrd




```
~/Projects/kindle/exploits/joyride  ‹master*› $ nc -lvp 44444
Listening on 0.0.0.0 44444
Connection received on 10.0.0.222 39203
id
uid=0(root) gid=0(root)
uname -a
Linux kindle 4.9.77-lab126 #1 SMP PREEMPT Thu Jul 24 00:11:45 UTC 2025 armv7l GNU/Linux
~/Projects/kindle/exploits/joyride  ‹master*› $
```

Epilogue

Epilogue

- **Timeline**
  - **01/12/25**: Report sent to Amazon Devices VRP
  - **10/12/25**: Triage confirmed high severity
  - **29/12/25**: Amazon raises severity to critical + $20k bounty
  - **12/01/26**: Fix released in production (5.19.2)

Epilogue

- **On the project**
  - It was a fun CTF challenge once I had the bugs
  - Feels good to jailbreak a device by yourself
  - Scratched my bootstrapping itch for now
- **On kindle security**
  - Latest kindles have better sandboxing and newer kernels (6.6)
  - But the LPE surface is still wild

Epilogue

- **Release**
  - https://github.com/synacktiv/kindle-overkill
  - Tested only on my devices (PW5 and PW4)
  - JS exploit could use some stabilization work
  - Not prod ready so use at your own risk !
- **Future work**
  - Who knows ?

Questions ?

SYNACKTIV

https://www.linkedin.com/company/synacktiv

https://x.com/synacktiv

https://bsky.app/profile/synacktiv.com

https://synacktiv.com