Leveling Up

How Hacking Consoles Launched My Cybersecurity Career

About me

- Security researcher with focus on low level security
- PlayStation console researcher since 16 years
  - Hacked PSP, PS Vita, PS4 and PS5

Level 12

- How it all started: Wanted a Nintendo DS to play Nintendogs 🐶
  - Got a PlayStation Portable instead
- The PSP was ahead of its time!
  - Spent all day and night playing (mostly demos)
  - Then started getting low grades in school :(
- Wanted to play games like GTA
  - Obviously not old enough
  - But also, no money 💸

Level 12

- My dad followed tutorials to crack the PSP using Pandora's Battery
  - Installed 4.01 M33 Custom Firmware
- Sony had a feature to boot into manufacturing mode via special bits set in the battery's eeprom.
  - Hackers reverse engineered and managed to exploit it to load unsigned code at boot
- 12yo me was fascinated by this witchcraft 🧙
  - First introduction to computer security
  - Motivated me to learn about computers and maths

Level 13

- Found tutorials to develop own homebrews in C.
  - For a couple of months, learned C by just staring at code
  - Collected all source codes from various homebrews and plugins.
- Because I believed I was a Noob, I named myself Total_Noob 🤓
- Eventually understood more and learned how to debug and use various tools to disassemble binaries.
  - Learned MIPS assembly.
  - Experimented with patching strings and hooking functions.

Level 14 - Level 16

- Learned about savedata exploits (stack buffer overflows) and found one myself.
- One day discovered an out-of-bounds write in a syscall.
- Exploitation was simple
  - Partial ASLR, no MMU (I didn't know about these concepts back then)
  - Just corrupt an instruction in kernel memory
- Reverse engineered M33 CFW by Dark AleX and created the first public Homebrew Enabler for newer firmwares and PSPgo
- Nobody believed it was real, coming from a 14yo guy with broken English 🤓

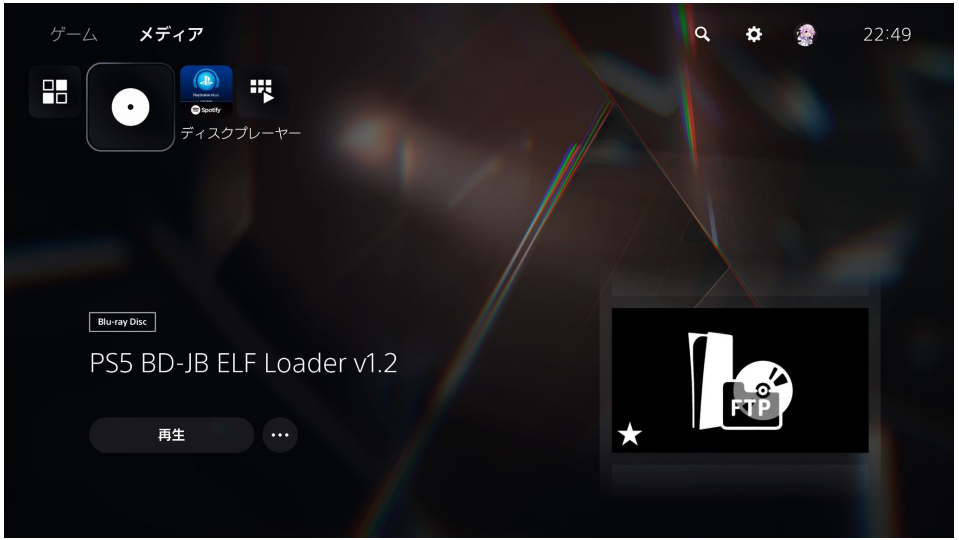
Level 17 - Level 20

- Spent a couple of years more doing PSP stuff for PS Vita.
  - PS Vita has a MIPS CPU on its SoC for PSP Emulation / backwards compatibility.
  - Exploited a couple more OOB write bugs.
  - Got very well acquainted with the PSP OS.
- However, I lacked modern OS / security experience outside of PSP / MIPS.

Level 21 - Level 23

- Pursued a BSc in CS
  - In free time, learned modern binary exploitation by working with the PS Vita.
- Created three jailbreaks for PS Vita:
  - **h-encore** – Savedata exploit with kernel exploit in ROP
  - **Trinity** – MIPS→ARM chain
  - **HENIo** – WebKit chain, still unpatched
- Covid project 😷: GTA:SA on Vita
  - Ported Android's ARMv7 .so game
  - Inspired 5+ people to create their own ports.

Level 24 - Level 29



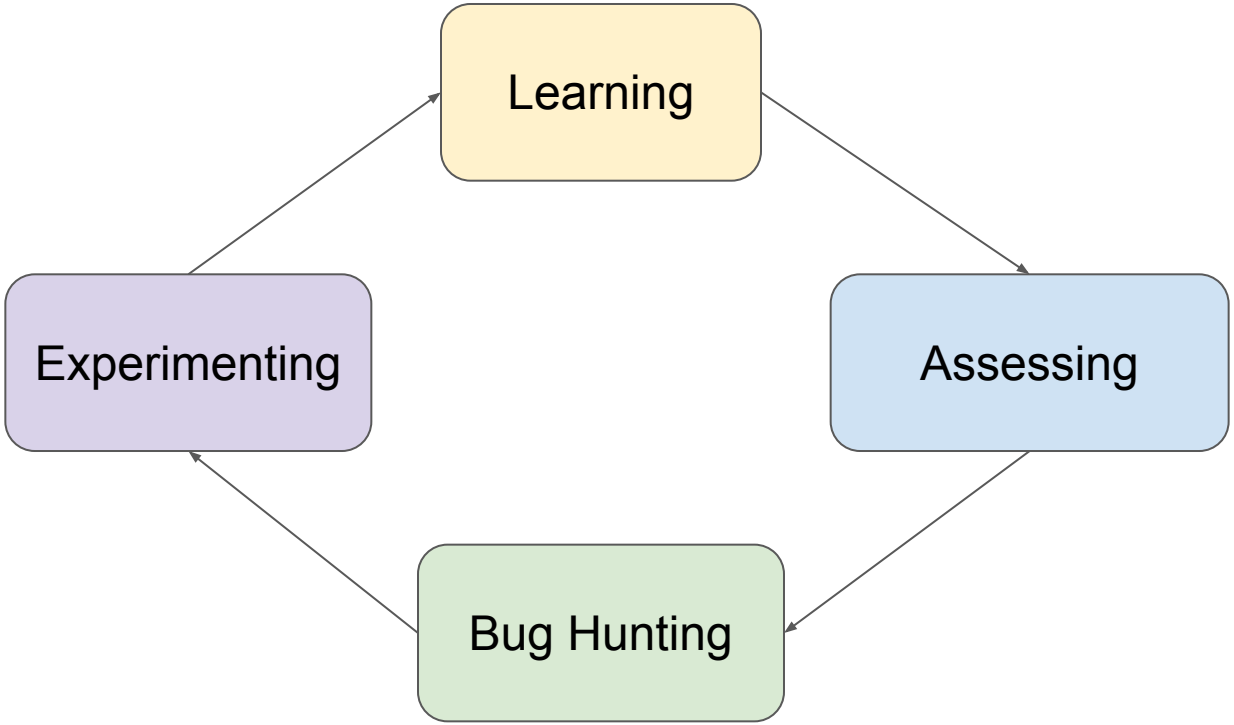
- Found, exploited and reported a couple of bugs for PS4/PS5
  - **IPv6 UaF** – Sony reintroduced bug in PS5 lol
  - **exFAT BoF** – requires plugging in a USB stick, so annoying
  - **PPPoE RCE** – takes 1min to trigger...
  - **bd-jb** – Blu-ray Disc Java Sandbox Escape – most popular jailbreaking method nowadays
- ETA WEN curse started 🤦

Level 24 - Level 29

- Applied to Google for internship after BSc
  - Got interviews for full-time job instead 😂
- Worked in a vulnerability management team
  - Got bored and that's why I researched PS4/PS5 on the side.
- Moved to Cloud Vulnerability Research (like Project Zero for Google Cloud)
  - Found lots of bugs, wrote lots of exploits
  - Simply applied knowledge from PlayStation hacking to Google servers
  - Got promoted 3x 🏆

Approaching Vulnerability Research

Vulnerability Research Lifecycle



Learning

- Get confident with all the bug classes in the system, e.g. for C/C++:
  - Integer overflow, Buffer-overflow, Use-after-free, Double-free, Type Confusion, etc.
  - Understand various root causes of such bugs.
- Find write-ups, academic papers, talks of prior or similar research
  - Try to understand as much as possible.
- Analyze public proof-of-concepts / exploits
  - Often, unit / regression tests include proof-of-concepts as well.
- Work on a non-security project related to the system to improve your understanding.
  - You may run into pitfalls that also happen to other developers.

Example: Console Research

- fail0verflow: https://fail0verflow.com/blog/
- CTurt PS4: https://cturt.github.io/
- Specter PS4: https://github.com/Cryptogenic/Exploit-Writeups
- oct0xor: https://github.com/oct0xor/presentations
- xyz PS Vita: https://blog.xyz.is/
- yifanlu PS Vita: https://yifan.lu/
- theflow PS Vita: https://theofficialflow.github.io/

Assessing

- Read source code / reverse engineer binaries
  - Understand logic / control flow
  - For console research: Understand boot process, e.g. what is the first instruction executed? What is the Root of Trust?
- Read documentation
  - Find relevant components in source code or binary
    Understand the threat model, and what privilege each component has
  - Take notes of interesting caveats and behaviors
- Browse through commits / look at “blame” layer
  - Understand what new features are being introduced, what kind of bugs they are fixing.
  - Oftentimes, vulnerabilities get fixed silently without any advisories.
- Figure out the configuration of the system
  - Also, understand how the code is compiled.

Assessing: Mapping the Attack Surface

- Enumerate ways to interact with the system
  - Find entry points where untrusted user data is processed
  - Identify interesting functions with names like recv, send, copy, parse, cmd, deserialize, etc.
- Enumerate all third-party projects used
  - Look for custom patches, configurations
- If existing, look at fuzzing harnesses
  - Indicate critical APIs
  - Identify areas with low coverage (high coverage != bug free)
- Try to find previously unknown attack surfaces
  - That's the gold mine

Bug Hunting

- Start with low hanging fruits, e.g. typical programming errors
  - Integer overflows such as malloc(a * b), or if (a + b < c)
  - Path traversals are very easy to find, and often quite impactful
  - Use static analysis tools such as weggli.
  - Or just grep / Ctrl+F for interesting functions
    - For heap vulnerabilities, go through all malloc / friends
    - For BoFs / OOBs, go through all memcpy's and array accesses
- Look for N-days in third-party components
- Look for variants of prior bugs
  - Devs usually just fix one particular bug reported to them.
- Eventually, look for architectural and logic bugs

Example: PPPwn Heap-buffer-overflow (CVE-2006-4304)

```
static int
sppp_lcp_RCR(struct sppp *sp, struct lcp_header *h, int len)
{
        // ...
        buf = r = malloc (len, M_TEMP, M_NOWAIT);
        // ...
        p = (void *)(h + 1);
        for (rlen=0; len>1 && p[1]; len-=p[1], p+=p[1]) {
                // ...
                /* Add the option to rejected list. */
                bcopy (p, r, p[1]);
                r += p[1];
                rlen += p[1];
        }
        if (rlen) {
                // ...
                sppp_cp_send(sp, PPP_IPCP, CONF_REJ, h->ident, rlen, buf);
                goto end;
        }
        // ...
}
```

Buffer for rejected options (controllable length)

Copy without checking the length

Example 2: FFmpeg Heap OOB Write (CVE-2022-2566)

```
sc->sample_offsets_count = 0;
for (uint32_t i = 0; i < sc->ctts_count; i++)
    sc->sample_offsets_count += sc->ctts_data[i].count;
av_freep(&sc->sample_offsets);
sc->sample_offsets = av_calloc(sc->sample_offsets_count, sizeof(*sc->sample_offsets));
...
for (uint32_t i = 0; i < sc->ctts_count; i++)
    for (int j = 0; j < sc->ctts_data[i].count; j++)
        sc->sample_offsets[k++] = sc->ctts_data[i].duration;
```

Integer overflow – count read with avio_rb32

Allocation too small

OOB write

Bug Hunting: Inconsistencies

- Figure out why certain things are done and what would happen if they were missing or done incorrectly
  - What sanity checks are made? How are they made? Why are they made?
  - Are there mutex locks? If so, then it means the code can run concurrently
  - Code is maintained by multiple people, therefore may not be consistent in quality.
- Discrepancies between design and implementation
  - Are there security properties that are violated?
- Parser differentials
  - Are things parsed in different ways? Interesting bugs could occur if so.
  - Simon Scannell found a very cool bug in CS:GO, where the HTTP header Content-Length was parsed in two different ways, leading to Uninitialized Heap Memory.

Example: FreeBSD IPv6 Use-after-free (CVE-2020-7457)

```
case IPV6_2292PKTOPTIONS:
{
        struct mbuf *m;

        // ...

        error = soopt_getm(sopt, &m); /* XXX */
        if (error != 0)
                break;
        error = soopt_mcopyin(sopt, m); /* XXX */
        if (error != 0)
                break;
        error = ip6_pcbopts(&inp->in6p_outputopts,
                            m, so, sopt);
        m_freem(m); /* XXX */
        break;
}
```

Called without a lock!

Example 2: VirtualBox OOB write (CVE-2023-22098)

```
static uint8_t virtioNetR3CtrlVlan(PVIRTIONET pThis, PVIRTIONET_CTRL_HDR_T pCtrlPktHdr, PVIRTQBUF pVirtqBuf)
{
    // ...
    AssertMsgReturn(uVlanId > VIRTIONET_MAX_VLAN_ID,
        ("%s VLAN ID out of range (VLAN ID=%u)\n", pThis->szInst, uVlanId), VIRTIONET_ERROR);

    // ...
    switch (pCtrlPktHdr->uCmd)
    {
        case VIRTIONET_CTRL_VLAN_ADD:
            ASMBitSet(pThis->aVlanFilter, uVlanId);
            break;
        case VIRTIONET_CTRL_VLAN_DEL:
            ASMBitClear(pThis->aVlanFilter, uVlanId);
            break;
        default:
            LogRelFunc(("Unrecognized VLAN subcommand in CTRL pkt from guest\n"));
            return VIRTIONET_ERROR;
    }
    return VIRTIONET_OK;
}
```

Wrong comparison used for assertion. Code didn't work!

OOB write

Bug Hunting: Assumptions

- Developers make assumptions
  - Usually, they assume the input is well-formed and forget to sanitize
  - Code changes over time. Are assumptions still valid?
- But also you make assumptions
  - Validate your assumptions.
  - Don’t assume a code is bug-free.
  - Don’t assume the developer thought about all corner cases, even if they have comments such as “// this is safe”
  - Don’t assume bugs are gone forever, sometimes regressions happen

Experimenting

- Don’t just stare at code, compile it and run it!
  - Compile the code with sanitizers
  - Compile the code with debug symbols and attach debuggers
  - Also useful to revert to a commit with known bugs
- Reproduce findings from other people
  - Rewrite to improve own understanding
  - Develop exploits to keep yourself motivated
- Write fuzzers / fuzzing harnesses
  - Attempt to find and trigger known bugs
- Write some simple proof-of-concepts to reach interesting code paths
  - Sometimes, rare code paths may not have been tested at all

Example: Bluetooth Type Confusion (CVE-2020-12351)

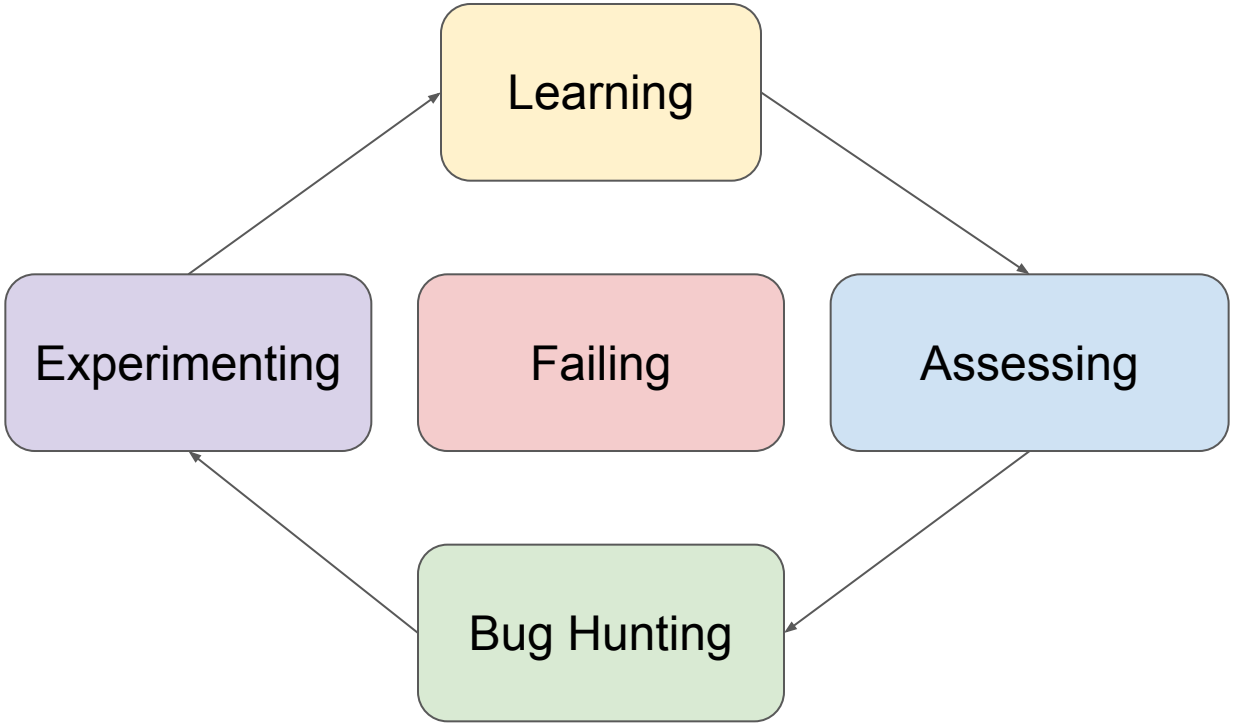
```
static struct amp_mgr *amp_mgr_create(struct l2cap_conn *conn, bool locked)
{
        struct amp_mgr *mgr;
        struct l2cap_chan *chan;
        ...
        chan = a2mp_chan_open(conn, locked);
        ...
        chan->data = mgr;
        ...
        return mgr;
}

static int l2cap_data_rcv(struct l2cap_chan *chan, struct sk_buff *skb)
{
        ...
        if ((chan->mode == L2CAP_MODE_ERTM ||
             chan->mode == L2CAP_MODE_STREAMING) && sk_filter(chan->data, skb))
                goto drop;
        ...
}
```

struct amp_mgr assigned to void *data

That expects a struct sock type!

Vulnerability Research Lifecycle



Failure and Motivation

- Be patient and persistent :)
- Always assume there are vulnerabilities
  - Imagine it's a CTF challenge
- Don't be discouraged by "almost-bugs"
  - Sometimes, from a different angle, bugs turn out to be exploitable
  - Or code changes and suddenly your bug becomes exploitable
- You may not find any bugs, but all the learning you've made is also valuable
- In case you're blocked, take breaks or switch targets
  - Easier targets → more results → feel good about yourself
- Find your fuel
  - **Challenge**: Product xyz is considered super secure
  - **Fame**: Be the first to pwn xyz
  - **Money**: Bounty for xyz is lots of $$$
  - **Free Games**: as 12yo kiddo, I simply wanted free games

Conclusion

- Be optimistic, patient and persistent.
- Iteratively learn, assess, hunt, experiment.
- Map the attack surface, start with low-hanging fruits, find inconsistencies, validate assumptions, and eventually look for architectural and logic bugs.
- Share your knowledge and information to teach and inspire the next generation of hackers.

Thanks for your attention and happy bug hunting!

Q&A