Penetration Test Report

NGICS - Felix86

V 1.0
Amsterdam, June 20th, 2026
Public

Document Properties

Client	NGICS - Felix86
Title	Penetration Test Report
Target	• Felix86 emulator
Version	1.0
Pentester	Spiros Thanasoulas
Authors	Spiros Thanasoulas, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	June 17th, 2026	Spiros Thanasoulas	Initial draft
0.2	June 19th, 2026	Marcus Bointon	Review
1.0	June 20th, 2026	Marcus Bointon	1.0

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between June 2, 2026 and June 17, 2026, Radically Open Security B.V. carried out a penetration test for NGICS - Felix86

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

Felix86 is an emulator that allows for the execution of x86/x86-64 binaries in RISC-V-hosted Linux kernels. It also packages a rootfs containing useful binaries and configuration files that would allow the emulated programs to have a familiar environment to execute in. Felix86 takes steps to contain the filesystem access of the emulated binary inside that rootfs, in order to protect the RISC-V system if malicious x86 binaries are executed.

1.2 Scope of Work

The scope of the penetration test was limited to the following target:

- Felix86 emulator

The scoped services are broken down as follows:

- Pentest and code review for the NGICS-Felix86 emulator.: 4 days
- Reporting.: 1 days
- **Total effort: 5 days**

1.3 Project Objectives

ROS will perform a penetration test of the Felix86 emulator in order to assess its security posture. To do so ROS will access the project's source code and guide its developers in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The security audit took place between June 2, 2026 and June 17, 2026.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 1 Extreme, 1 High, 2 Moderate and 3 Low-severity issues.

We discovered issues ranging from filesystem sandbox escape, to memory corruption, code injection, and denial of service. The most severe finding is a Github Actions pipeline shell injection (page 12) that can lead to exposed CI secrets and supply chain tampering, followed by a complete rootfs sandbox escape (page 14) when a directory file descriptor is opened before a guest `chroot()` call, survives the call, and is later used as a traversal anchor. This can allow a guest program to walk up and straight out to the real host filesystem. A third high-severity finding involves use of unbounded string sprint/sscanf operations (page 13) on stack-allocated buffers. Attacker-controlled input can reach these buffers via instrumented directories present in the guest's mountinfo, and the resulting corruption allows for an attacker to run shellcode with the privilege of the user that invoked Felix86.

Additional findings include a system()-based command injection (page 15) reachable through unsanitized executable paths when prompting the user, and a denial of service via unbounded alloca sizing (page 17) in the epoll syscall handlers. In addition Felix86 hardcodes `AT_SECURE` (page 18) and the guest's `uid/euid/gid/egid` to fixed, always-"unprivileged" values when building the initial process stack, which permanently disables the dynamic linker's protection against `LD_PRELOAD/LD_LIBRARY_PATH` injection. Another low severity finding is a predictable, world-writable log `FIFO` (page 16) that allows for log tampering and denial of service.

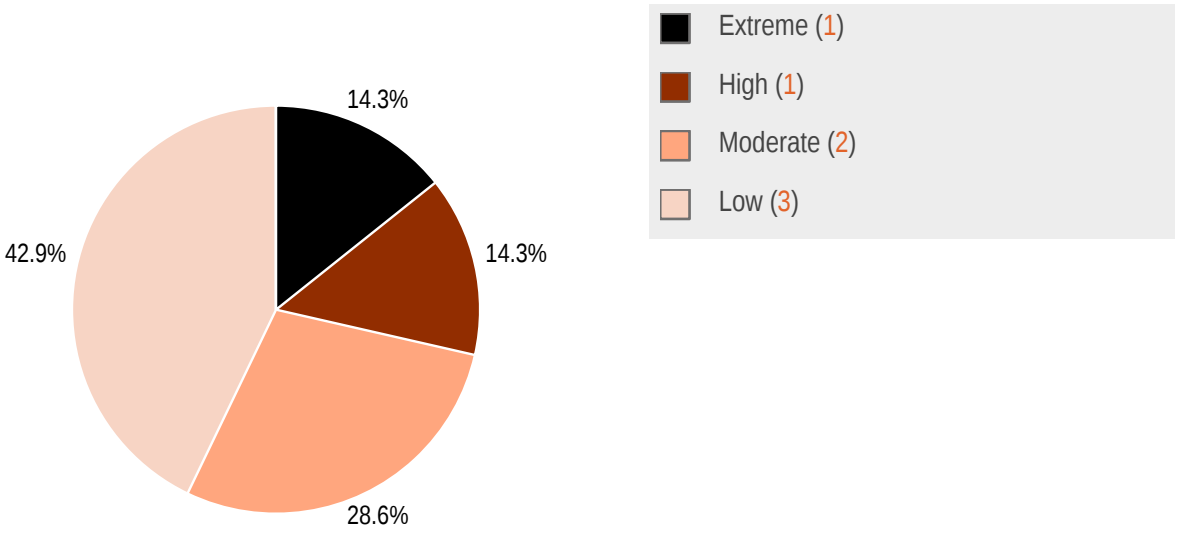
Finally, we worked with the development team in suggesting an improved approach for the handling of secrets needed in the rootfs, for the authentication of `SUID` programs that the user might want to execute under Felix86.

1.6 Summary of Findings

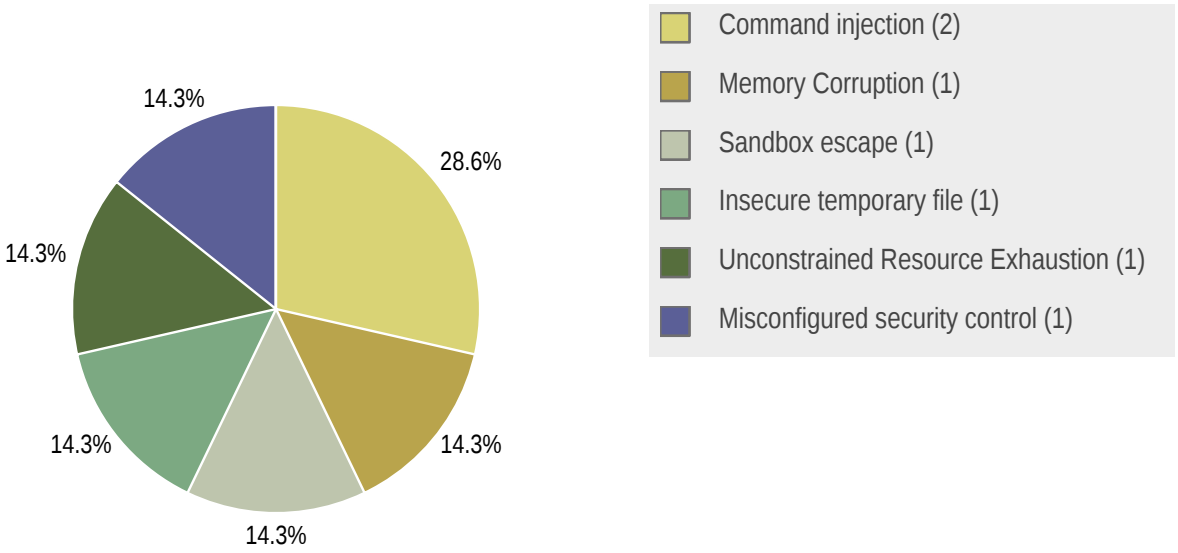
Info	Description
CLN-002 **Extreme** **Type:** Command injection **Status:** none	build.yml interpolates ${{ github.ref }} directly into a run: shell block, permitting shell injection.
CLN-006 **High** **Type:** Memory Corruption **Status:** none	Multiple unbounded string write operations allow for stack corruption in utility.cpp.
CLN-001 **Moderate** **Type:** Sandbox escape **Status:** none	Guest binaries in trusted folders can escape rootfs containment by saving a directory file descriptor, calling chroot to corrupt the escape-detection anchor, then using openat with the saved descriptor to traverse the host filesystem freely.
CLN-005 **Moderate** **Type:** Command injection **Status:** none	Attacker controlled input is injected in the command line passed to system.

CLN-003 **Low** **Type:** Insecure temporary file **Status:** none	The log pipe file is created with world writable permissions and with a predictable name.
CLN-004 **Low** **Type:** Unconstrained Resource Exhaustion **Status:** none	A guest controlled value of maxevents can cause the emulator to attempt multi gigabyte allocations.
CLN-007 **Low** **Type:** Misconfigured security control **Status:** none	Felix86 hardcodes AT_SECURE to 0 and AT_UID/AT_EUID/AT_GID/AT_EGID to default values for every guest binary, regardless of the binary's actual privilege state.

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

Info	Recommendation
CLN-002 **Extreme** **Type:** Command injection **Status:** none	• Pass values through an intermediate `env:` variable, and reference them in the shell as a quoted string.
CLN-006 **High** **Type:** Memory Corruption **Status:** none	• Add field-width limits to the `sscanf` format string (e.g. `%511s`), replace `sprintf` with `snprintf` and check the return value against the buffer size. • Alternatively remove the fixed `char[]` buffers entirely in favor of `std::string`-based parsing and `fmt::format` (already used in other parts of the codebase), which eliminates the overflow class rather than just bounding it.
CLN-001 **Moderate** **Type:** Sandbox escape **Status:** none	• Re-validate that a passed-in fd is still inside the current rootfs before using it as a resolution anchor.
CLN-005 **Moderate** **Type:** Command injection **Status:** none	• Avoid using `system`(); use `execve`() or `posix_spawn`() instead.
CLN-003 **Low** **Type:** Insecure temporary file **Status:** none	• Restrict FIFO permissions. • Use unguessable filenames for temporary files.

CLN-004 **Low** **Type:** Unconstrained Resource Exhaustion **Status:** none	• Add bounds checking for arg3 in the four `alloca` calls.
CLN-007 **Low** **Type:** Misconfigured security control **Status:** none	• `stat`() the binary before execution and propagate the correct values for the auxiliary vector.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**
   We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.
2. **Enumeration**
   We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.
3. **Scanning**
   Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.
4. **Obtaining Access**
   We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately though provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2021) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Reconnaissance and Fingerprinting

We were able to quickly discover areas of interest in the code base using the following industry-standard tools:

- semgrep – Pattern based static analysis
- CodeQL – Data flow and taint analysis tool
- CppCheck – C++ focused static analyzer
- Qwen2.5-7B – Local LLM assisting in triaging of potential findings

4 Findings

We have identified the following issues:

4.1 CLN-002 — GitHub runner is vulnerable to shell injection via github.ref interpolation

Vulnerability ID: CLN-002

Vulnerability type: Command injection

Threat level: Extreme

Description:

`build.yml` interpolates `${{ github.ref }}` directly into a `run:` shell block, permitting shell injection.

Technical description:

As can be seen in `build.yml`:

```
32        run: |
33          EXTRA_FLAGS=""
34          if [[ "${{ github.ref }}" == refs/tags/* ]]; then
35            EXTRA_FLAGS="-DFELIX86_MONTHLY_RELEASE=1"
36          fi
```

the markup interpolates `${{ github.ref }}` directly into a `run:` shell block. Any commands injected into `github.ref` will be executed by the CI/CD pipeline.

Impact:

An attacker controlling the values passed into `github.ref` can execute arbitrary code, leaking credentials or infecting the repo artifacts with malware.

Recommendation:

Pass values through an intermediate `env:` variable, and reference them in the shell as a quoted string `("$EXTRA_FLAGS")`, which the shell treats as data rather than syntax. Apply this pattern to all uses of `github.ref`/`github.head_ref` etc.

4.2 CLN-006 — Misuse of the sscanf/sprintf APIs can lead to stack buffer overflows

Vulnerability ID: CLN-006

Vulnerability type: Memory Corruption

Threat level: High

Description:

Multiple unbounded string write operations allow for stack corruption in utility.cpp.

Technical description:

`felix86_mountinfo()` in `common/utility.cpp` parses `/proc/self/mountinfo` to rewrite mount paths before exposing them to the guest. This makes the rootfs mounts look sane to the emulated environment. Each line is parsed with `sscanf(..., "%s %s", devid, path1, path2)` into fixed 512-byte stack buffers (`devid, path1, path2`) with no field-width limit, so any `mountinfo` field longer than 511 bytes overflows the stack. The parsed fields are then reassembled with `sprintf` into 4096-byte stack buffers (original, replacement) with no bound on the combined output length. The replacement variable is built from `path1s`/`path2s`, `std::strings` that can grow arbitrarily (e.g. if mount path substitution lengthens rather than shortens the string), so even moderately long original paths can produce a formatted string exceeding 4096 bytes.

Below we list the problematic code paths in `common/utility.cpp`:

```
1615            // what happens here is brain dead
1616            // do the magic here
1617            char path1[512];
1618            char path2[512];
1619            int nid2;
1620            char devid[512];
1621
1622            std::string ori = line;
1623
1624            sscanf(line.c_str(), "%d %d %s %s %s", &nid, &nid2, devid, path1, path2);
1625
1626            char original[4096];
1627            sprintf(original, "%d %d %s %s %s", nid, nid2, devid, path1, path2);
...
1636            char replacement[4096];
1637            sprintf(replacement, "%d %d %s %s %s", nid, nid2, devid, path1s.c_str(),
 path2s.c_str());
```

Impact:

An attacker managing to corrupt the stack via these buffer overflows, can gain code execution outside the emulated environment with the privileges of the user executing Felix86.

Recommendation:

- Add field-width limits to the `sscanf` format string (e.g. `%511s`), replace `sprintf` with `snprintf` and check the return value against the buffer size.
- Alternatively remove the fixed `char[]` buffers entirely in favor of `std::string`-based parsing and `fmt::format` (already used in other parts of the codebase), which eliminates the overflow class rather than just bounding it.

4.3 CLN-001 — Rootfs sandbox escape via chroot

Vulnerability ID: CLN-001

Vulnerability type: Sandbox escape

Threat level: Moderate

Description:

Guest binaries in trusted folders can escape rootfs containment by saving a directory file descriptor, calling chroot to corrupt the escape-detection anchor, then using openat with the saved descriptor to traverse the host filesystem freely.

Technical description:

`resolveImpl` in `filesystem.cpp` enforces containment by walking `..` components and checking each intermediate directory's inode against `root_statx`, which is computed from `g_rootfs_fd` at call time. When a relative path is resolved against a passed-in fd, `current_fd` is set directly to that fd with no check that it's still inside the current rootfs. Consequently, a guest can:

- Open a directory fd while inside a trusted folder's host path
- Call `chroot()` to a subdirectory inside the rootfs, which repoints `g_rootfs_fd`/`root_statx` to a further nested inode
- Call `openat(saved_fd, "../../../../../../etc/hostname")`

The `..` walk from `saved_fd` never encounters the new, much-deeper `root_statx`, so every escape check passes, and the open call succeeds against the real host filesystem.

Impact:

Arbitrary host filesystem read/write outside the intended rootfs sandbox, from inside the guest.

Recommendation:

- Re-validate that a passed-in fd is still inside the current rootfs before using it as a resolution anchor.

4.4 CLN-005 — Shell command execution via system

Vulnerability ID: CLN-005

Vulnerability type: Command injection

Threat level: Moderate

Description:

Attacker controlled input is injected in the command line passed to `system`.

Technical description:

`main.cpp` passes an executable path into `system`() to invoke `whiptail`/`zenity` for a GUI prompt when the target binary lives outside the rootfs. This can be seen below

```
651         if (std::filesystem::exists("/bin/whiptail")) {
652             status = WEXITSTATUS(
653                 system(("/bin/whiptail --title \"felix86: Add to trusted folders?\" --yes-button
 Yes --no-button No --yesno \"" +
654                         (canonical_path.string() + " seems to be outside the rootfs." + " Would
 you like to add the parent folder " +
655                          parent.string() + " to the trusted folders?") +
656                         "\" 0 0")
657                            .c_str()));
658         } else {
659             status = 2;
660             WARN("Couldn't find /bin/whiptail to ask user if they want to trust the folder");
661         }
```

The path isn't sanitized, and `system()` executes it by invoking a shell.

Impact:

Arbitrary command execution as the user running Felix86, triggered by controlling the path of a binary Felix86 is asked to execute (e.g. via a crafted filename, symlink, or argument). Practical payloads are constrained to commands that do not contain `/`, but that still allows meaningful actions (env var dumps, network callbacks via tools already on `PATH`, writing files via redirection).

Recommendation:

- Avoid using `system`(); use `execve`() or `posix_spawn`() instead.

4.5 CLN-003 — Predictable world writable log pipe

Vulnerability ID: CLN-003

Vulnerability type: Insecure temporary file

Threat level: Low

Description:

The log pipe file is created with world writable permissions and with a predictable name.

Technical description:

`Logger::startServer` creates a FIFO at a predictable path `/tmp/felix86-{pid}.pipe` and explicitly sets it world-writable as can be seen in `common/log.cpp`

```
25    std::string log_path = "/tmp/felix86-" + std::to_string(getpid());
26    g_pipe_name = log_path + ".pipe";
27    log_path += "-XXXXXX.log";
28    int fd = mkstemps(log_path.data(), 4);
29    ASSERT(fd != -1);
30
31    int ok = mkfifo(g_pipe_name.c_str(), 0666);
32    ASSERT(ok == 0);
33
34    // mkfifo uses umask to set the permissions, override them
35    ok = chmod(g_pipe_name.c_str(), 0666);
36    ASSERT(ok == 0);
```

Impact:

A local attacker can pre-create this path as a symlink to an arbitrary location before Felix86 starts, causing mkfifo to fail or the pipe to be created elsewhere. In addition, since the pipe is world-writable after creation, any local user can manipulate logging, or hold the read end open to block Felix86's logging thread.

Recommendation:

Create the FIFO with restrictive permissions (`0600`) instead of `0666`, and avoid the predictable `/tmp/felix86-{pid}` naming pattern. Use mkstemp-style randomized naming or place it in a per-user, non-world-writable directory (e.g. under $XDG_RUNTIME_DIR).

4.6 CLN-004 — DoS via stack exhaustion in epoll handler

Vulnerability ID: CLN-004

Vulnerability type: Unconstrained Resource Exhaustion

Threat level: Low

Description:

A guest controlled value of maxevents can cause the emulator to attempt multi gigabyte allocations.

Technical description:

The `epoll_wait`, `epoll_pwait`, and `epoll_pwait2` syscall handlers allocate a host stack buffer with `alloca`, sized directly from the guest-controlled `maxevents` argument with no upper-bound check, as can be seen in `hle/syscall.cpp`:

```
1806         case felix86_x86_64_epoll_wait: {
1807             epoll_event* host_events = (epoll_event*)alloca(std::max(0, (int)arg3) *
 sizeof(epoll_event));
1808             result = epoll_wait((int)arg1, host_events, (int)arg3, (int)arg4);
```

Impact:

A guest binary can crash the Felix86 host process by calling `epoll_wait`/`epoll_pwait` with an oversized `maxevents` value, terminating all guest threads running under that Felix86 instance.

Recommendation:

- Add bounds checking for arg3 in the four `alloca` calls.

4.7 CLN-007 — Hardcoded AT_SECURE and credential auxv disable runtime linker protections

Vulnerability ID: CLN-007

Vulnerability type: Misconfigured security control

Threat level: Low

Description:

Felix86 hardcodes `AT_SECURE` to 0 and `AT_UID/AT_EUID/AT_GID/AT_EGID` to default values for every guest binary, regardless of the binary's actual privilege state.

Technical description:

The auxiliary vector entries are initialized as follows in `emulator.cpp`:

```
     123 std::vector<std::pair<u64, u64>> auxv_entries = {
124         {AT_PAGESZ, {4096}},
125         {AT_EXECFN, {(u64)program_name}},
126         {AT_CLKTCK, {100}},
127         {AT_ENTRY, {elf->GetEntrypoint()}},
128         {AT_PLATFORM, {(u64)platform_name}},
129         {AT_BASE, {(u64)elf->GetProgramBase()}},
130         {AT_FLAGS, {0}},
131         {AT_UID, {1000}},
132         {AT_EUID, {1000}},
133         {AT_GID, {1000}},
134         {AT_EGID, {1000}},
135         {AT_SECURE, {0}},
136         {AT_PHDR, {(u64)elf->GetPhdr()}},
137         {AT_PHENT, {elf->GetPhent()}},
138         {AT_PHNUM, {elf->GetPhnum()}},
139         {AT_RANDOM, {rand_address}},
140         {AT_HWCAP, {0xBFEBFBFF}},
141     };
```

These values are supposed to be unique for every binary executed by the linker, and are supposed to be populated via `stat()`-ing the executable at hand. The auxiliary vector is supposed to communicate information to the dynamic linker, in order to enforce certain protections (like disabling the loading of shared libraries via `LD_PRELOAD` and more).

Hardcoding these values doesn't allow, for example, the linker to determine appropriately whether this binary is a root owned `SUID` that would need to set `AT_SECURE` to 1 during its execution.

Impact:

An unprivileged process inside the Felix86 rootfs, with the ability to write a shared library to disk, can then `LD_PRELOAD` it on a root-owned setuid executable, therefore executing arbitrary code after it has elevated privilege.

Recommendation:

- `stat`() the binary before execution and propagate the correct values for the auxiliary vector.

5 Future Work

- **Credential handling inside the rootfs**
  Currently Felix86 copies the host credential files (passwd/shadow) into the rootfs in order to facilitate the user experience of a user performing a privileged task via sudo. This decision negatively impacts the security posture of the emulator, since an attacker gaining filesystem access will initially be contained inside the rootfs. Exposing actual system secrets in there, can give the attacker access to the host. There are two other ways for Felix86 to deal with password based authentication in the emulated environment: using default credentials or issuing randomized credentials. The former, combined with the ability of the emulated system to gain superuser privileges on the host kernel, can lead to complete system compromise. This can also allow worm-able payloads, where if it is detected that a payload is executing under Felix86, directly via the leverage of the default credential situation, malware can elevate to superuser on the host system. Therefore, only the latter approach is a sensible solution to the problem. Randomized credentials should be generated and their hashes stored (the plaintext should be shown only once to the user and not logged). However, we must stress that our recommendation for future versions of Felix86 would be to opt for a default where elevated credentials are not allowed in the emulated environment at all.
- **Retest of findings**
  When mitigations for the vulnerabilities described in this report have been deployed, perform a repeat test to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

6 Conclusion

We discovered 1 Extreme, 1 High, 2 Moderate and 3 Low-severity issues during this penetration test.

During this assessment, we worked closely with the Felix86 development team to identify and address the presented issues. Throughout the engagement the developers were extremely responsive and showed a genuine commitment to resolving findings quickly and thoroughly. Felix86 takes steps to protect the hosting system from misbehaving emulated programs, but is also careful to point out in the documentation that these features should not be relied upon as security controls. At the moment the purpose of the emulator is to provide a frictionless way for RISC-V users to run userspace x86 applications (games/multimedia). We worked with the development team to provide a comprehensive road map for tightening the default settings in future versions without sacrificing the user experience.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process that must be continuously evaluated and improved – this penetration test is just a one-time snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing Team

Spiros Thanasoulas *(pentester)*	Spiros is a freelance security engineer, focusing on operating systems and networks. His interests include vulnerability research, exploit development and CTF competitions. He holds a BSc in Physics from the University of Crete and an MSc in Computer Science from the University of Illinois - Urbana Champaign
Melanie Rieback *(approver)*	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.