Security Research Labs

Reproducible Builds
Baseline Security Assurance

Threat model and hacking assessment report

v1.0 FINAL, 14 January 2025

Prepared for:
Reproducible Builds

- 23 pages -

Content

Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the timeframe and scope detailed in chapter 3.

Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively and that following all evolution suggestions described in chapter 2 may not ensure all future code to be bug free.

Version:	v1.0 FINAL	
Prepared For:	Reproducible Builds	
Date:	14 January 2025	
Prepared By:	Daniel Schmidt	schmidt@srlabs.de
	Marc Heuse	marc@srlabs.de
	Florian Wilkens	florian@srlabs.de

Timeline

The Reproducible Builds source code security assessment started end of October 2024 and the analysis took 11 weeks.

Date	Event
October 29, 2024	Project Kickoff
December 23, 2024	Report for the baseline security check delivered
January 14, 2025	Final Report delivered (this document)

Table 1: Audit timeline

1 Executive summary

Reproducible Builds is a set of software tools and practices that enable projects to ensure their software builds are verifiably reproducible. This means that the build process produces a deterministic output (i.e., build artifacts) from the same source code, given the same build environment. By achieving reproducibility, projects can confidently verify that the built artifacts have not been tampered with, enhancing security and trust. Additionally, reproducibility enables more efficient caching of build artifacts and facilitates reproducing bugs.

1.1 Engagement overview

This report documents the results of the security assurance audit performed by Security Research Labs on three tools developed by the Reproducible Builds project:

[1] diffoscope
[2] strip-nondeterminism
[3] reprotest

Security Research Labs is a consulting think tank that provides specialized audit services in the security ecosystem since 2010.

During this study, the Reproducible Builds team provided access to relevant documentation and effectively supported the assessment. We verified the architecture, concept documentation, and relevant available source code of the three tools in scope from Reproducible Builds.

This audit focused on assessing the codebase for resilience against hacking and abuse scenarios. Key areas of scrutiny included the differential report formats, common client web attacks, command injections, privilege management, data leakage, hiding of modifications in build process and possible attack vectors to enable denial of service against the build process. The testing approach combined manual code inspection and static analysis. We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of Reproducible Builds' tools. We collaborated closely with the Reproducible Builds' team, utilizing full access to source code and documentation to perform a rigorous assessment.

1.2 Observations and Risk

We identified several issues ranging from medium to informational severity. No high or critical issues were identified during the audit. Reproducible Builds acknowledged all reported issues.

1.3 Recommendations

In addition to mitigating the issues, we recommend integrating comprehensive security testing, such as fuzz testing and security focused end-to-end tests.

Additionally, we recommend applying an "isolated by default" policy, which prevents users from accidentally running the Reproducible Builds tools without isolation unless they explicitly choose to do so.

Furthermore, we recommend more explicit dependency management to avoid unknowing use of vulnerable dependencies.

Finally, we recommend conducting iterative threat modeling and applying additional secure development best practices to help identify potential risks early on and ensure the integrity of the Reproducible Builds tools.

2 Evolution suggestions

We are pleased to report that the Reproducible Builds' security measures are sufficiently robust and align with established industry standards. To ensure that Reproducible Builds is secure against further unknown or yet undiscovered threats, we recommend considering the evolution suggestions and best practices described in this section.

2.1 Business logic improvement suggestions

Restrict capabilities on the host. The diffoscope tool inherently needs to deal with a high number of different file formats. This poses a broad attack surface as file format parsing is a common target of exploitation. The tool reprotest is exposed to similar risks as it directly executes build commands, which essentially amounts to unrestricted code execution. To reduce these risks, the tools should ´be restricted in their capabilities on the host system beyond their required permissions. Common approaches to achieve this are containerization, chrooting, or rewriting the code to explicitly drop privileges after initialization. While the documentation already recommends isolation as best practice, the tools currently do not take any steps to verify that this is indeed the case.

We recommend extending diffoscope, reprotest and strip-nondeterminism to perform isolation detection and refuse operation if no adequate isolation is found. Additionally, an `--insecure` flag should be introduced to manually accept the risks of missing isolation in cases where it cannot be provided. In this mode the tools should still issue a warning to the user highlighting the exploitation risks. In the long-term, the tools could also be rearchitected to drop capabilities after initialization, for example, via seccomp. This would reduce the need for external isolation as the attack surface is reduced in the tools themselves.

Clearly manage and version dependencies. Diffoscope currently specifies only a minimal set of required dependencies directly in `setup.py`. This ensures that it can be installed on a large set of machines which might not have a multitude of optional dependencies installed. However, the optional dependencies, especially in the `comparators` extras section, are not clearly advertised in the README and sometimes offer significant security improvements over the fallback options, for example in the case of `defusedxml` over the fallback implementation based on the `xml.minidom` module from Python standard library. Additionally, diffoscope does not inform the user once fallback implementations are used, outside of a dedicated `--list-missing-tools` argument which does not perform normal operation and is likely to be missed.

Another dependency-related issue lies in the absence of version specifiers for all dependencies across the two audited Python tools. Without version restrictions in place, a package installer like `pip` may rely on outdated and potentially vulnerable dependencies that were installed in the Python environment or globally in the past, even though newer and more secure versions are available. Adding the version specifier shifts some responsibility of dependency management from the end user to the tool authors and improves the default behavior for non-expert users as the

We recommend (1) specifying minimum version requirements for both required and optional dependencies across all three tools with regular updates when vulnerabilities are found and fixed upstream and (2) advertising the optional dependencies, especially for diffoscope, more directly in the README in a way that mirrors the existing section about optional external tools, while also printing a message to the user if fallback implementations are used or ideally even dropping potentially insecure fallback options.

2.2 Secure development improvement suggestions

We recommend further strengthening the security of the Reproducible Builds tools by implementing the following recommendations:

Perform threat modeling. Threat modeling for all new features and major updates before coding promotes better code security. This practice allows developers to identify potential security threats and vulnerabilities early in the design phase and implement appropriate mitigations right from the start. Including the threat model in the pull request description ensures that the entire team is aware of the identified risks and the measures taken to address them, promoting a proactive security culture and enhancing the overall robustness of the codebase. Additionally, it helps the audit team to identify gaps in the threat model and focus their assessment.

Leverage static analysis tooling. Static analysis tools help detect security flaws in the codebase, thus improving code security. These tools, such as `bandit` [4] or `safety` [5], analyze code without executing it. They identify vulnerabilities, coding errors, and compliance issues early in the development process. This proactive approach helps developers address potential security issues before they reach production, ensuring a more secure and reliable codebase.

Perform dynamic analysis. Developing fuzzing harnesses for different file format comparisons, output formats and other critical components is essential for identifying security vulnerabilities and business logic issues. By employing invariants, these fuzzing tests can effectively uncover subtle flaws that might otherwise go unnoticed. This demonstrates how comprehensive and targeted fuzz testing can significantly enhance the security and reliability of complex systems.

2.3 Address currently open security issues

We recommend addressing already known security issues once time permits. Even if an open issue has a limited impact, an attacker might use it as part of their exploitation chain, which may have a more severe impact on the Reproducible Builds project.

2.4 Further recommended best practices

End-to-end test implementation. Although the in-scope Python tools contain unit tests that largely cover their respective functionality, they only check the tools' functionality in isolation and do not consider the entirety of the reproducibility process: Potential problems in the tools' interaction can remain undetected. We recommend implementing end-to-end test cases, e.g., via reproducible or explicitly non-reproducible dummy packages that explicitly introduce errors in certain steps of the reproducibility pipeline. These end-to-end tests ensure that the overall process of reproducible builds works as expected even when failures occur in the interaction between tools.

Additionally, we recommend extending unit tests to provide a stronger focus on security. This includes testing for crashes in the used third-party tools, edge cases, potential vulnerabilities, and common attack vectors. Furthermore, tracking of test coverage metrics can help to identify blind spots and untested code.

Documentation. The documentation occasionally lacks explanations of the internal code architecture. This complicates understanding the codebase and diverts valuable time from verifying the tool's functionality. This is also especially relevant outside of (security) audit contexts, e.g., when new developers need to be onboarded to the codebase for feature development. To improve the architecture documentation, establish a practice of updating design documents and code comments concurrently with any code changes or new features. Incorporating documentation verification into the code review process can help detect discrepancies early.

3 Motivation and scope

The Reproducible Builds project aims to detect and prevent supply-chain attacks on binary packages for Linux distributions by offering independently verifiable artifacts that can be used to determine if a binary package has been tampered with during the build process. This is achieved by (1) pushing software towards being reproducible, i.e., making it produce bit-by-bit identical copies in identical build environments and (2) running independent build servers that build distribution packages and can be used as base for comparison with official upstream packages.

Security Research Labs collaborated with the Reproducible Builds team to create an overview containing three key software tools in scope and their audit priority. The in-scope components and their assigned priorities are reflected in table 2. During the audit, we used threat modelling to guide our efforts on exploring potential security flaws and realistic attack scenarios.

Tool	Priority	Description
diffoscope [1]	High	Python application to obtain human-readable diffs between files or folders supporting many file formats.
strip-nondeterminism [2]	Medium	Perl program to strip non-deterministic information such as timestamps from various file and archive formats.
reprotest [3]	Low	Python application to build source code multiple times in varying build environments to check reproducibility.

Table 2: Tools from Reproducible Builds that are in scope with audit priority

The three tools were chosen as they represent critical components of the overall reproducibility process:

- **diffoscope** is a Python application that produces detailed diffs between files or between directories. In the reproducibility process it is used to detect changes in the build artifacts introduced by build system variations (e.g., if the package is not properly reproducible). To enable meaningful diffs and support the developer in addressing potential issues, it parses various file formats common in software packages such as archives and binary formats. It is also used outside of Reproducible Builds, e.g., when comparing binary packages of propiertary software after updates or for context-aware diffs between different versions of pdf documents.

  The audit of diffoscope focused on the parts of the codebase that directly interact with the various supported file formats, as they pose an inherent risk for potential exploitation due to the complex parsing logic for certain file formats. The relevant classed are located in the `diffoscope.comparators` package.

- **strip-nondeterminism** is a Perl script that removes non-deterministic elements from various kinds of files such as timestamps or ordering differences in archives. This tool is essential for build pipelines that produce reproducible packages. It eliminates non-deterministic elements often introduced during post-processing or packaging, rather than during the actual compilation of the source code. As such, it also supports various formats commonly used in software packaging similar to diffoscope. The strip-nondeterminisim command is usually called last in a build pipeline and the resulting package is treated as the final build artifact that is reproducibly buildable from other machines with the same build environment.

  The audit of strip-nondeterminism focused on common Perl best-practices especially related to file handling and interaction with spawned processes as well as the file format parsing as they pose the largest risk of exploitation from malformed input files.

- **reprotest** is a Python application that builds a source repository in varying build environments and compares the resulting packages for differences via diffoscope or another fallback diff tool. Its main purpose is running on dedicated build servers maintained by the Reproducible Builds team to offer signed testimonials that a reproducible build was performed. In the future, these signatures and the respective binary packages can then be compared with the version served by distribution package managers to verify that those have not been tampered with.

  The reprotest audit focused on the parts of the codebase that directly interact with the build environment (the so-called `testbed` in the code) as well as security best practices and results of static analysis tools.

4 Methodology

We applied the following four-step methodology when performing the security assessment for the Reproducible Builds tools:
(1) threat modeling,
(2) security design coverage checks,
(3) implementation baseline check, and finally
(4) remediation support.

4.1 Threat modeling and attacks

The threat model framework's goal is to determine specific areas of risk in the Reproducible Builds tools. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios.

The risk level is categorized into low, medium, and high; considering both the hacking value and the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking\ Value = \frac{Damage \times Incentive}{Effort}$$

The *Hacking Value* is similarly categorized into low, medium, and high, and considers the incentive of an attacker, as well as the effort required by an adversary to successfully execute the attack. The hacking value is calculated as follows:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While incentive describes what an adversary might gain from performing an attack successfully, effort estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- **Low:** Attacks offer the hacker little to no gain from executing the threat
- **Medium:** Attacks offer the hacker considerable gains from executing the threat
- **High:** Attacks offer the hacker high gains by executing this threat

Effort:

- **Low:** Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources
- **Medium:** Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge
- **High:** Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors

Incentive and Effort are divided according to table 3.

	Incentive		
Effort	**Low incentive**	**Medium Incentive**	**High Incentive**
High effort	Low	Medium	Medium
Medium effort	Medium	Medium	High
Low effort	Medium	High	High

Table 3: Hacking value is determined by effort and incentive

Hacking scenarios are classified by the risk they pose to the system. Conversely, the *Damage* describes the negative impact that a given attack, if performed successfully, would have on the victim. The degrees of damage are defined as follows:

Damage:

- **Low:** Risk scenarios would cause negligible damage to the Reproducible Builds process
- **Medium:** Risk scenarios pose a considerable threat to the Reproducible Builds process
- **High:** Risk scenarios pose an existential threat to the Reproducible Builds process

Damage and Hacking Value are divided according to table 4.

	Hacking value		
Damage	**Low hacking value**	**Medium hacking value**	**High hacking value**
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 4: Risk is determined by damage and hacking value

After applying the framework to the Reproducible Builds ecosystem, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, and availability.

Confidentiality: Confidentiality threat scenarios involve sensitive information related to systems running the Reproducible Builds tools, primarily concerning data on the host system. Examples include attackers exploiting information leaks on a build server, such as SSH keys, to compromise the host machine.

Integrity: Integrity threat scenarios aim to disrupt the functionality of the Reproducible Builds process by undermining or bypassing the reproducibility of an application that uses Reproducible Builds tools.

This includes, for example, scenarios where modifications to the application remain undetected in the Reproducible Builds pipeline. Undermining the integrity of the Reproducible Builds process often comes with a high monetary incentive. For instance, if an attacker can integrate undetected changes into a package, they could include a backdoor, potentially infecting all users of the application.

Availability: Availability threat scenarios involve compromising the availability of the build pipeline used by projects utilizing Reproducible Builds tools, as well as the availability of the host systems executing these tools. Key threat scenarios include denial-of-service (DoS) attacks on the build pipeline and misleading reproducible tools into falsely believing a project is not reproducible, potentially delaying the release of a package.

Table 5 provides a high-level overview of the hacking risks associated with the identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them is provided in the threat model deliverable. This list can serve as a starting point for the Reproducible Builds developers to guide their security outlook for future feature implementations. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

Security goal	Hacking value	Example threat scenario	Easiness of attack	Example attack ideas
Confidentiality	High	A malicious package exploits a command injection vulnerability in diffoscope, leading to the disclosure of sensitive information on the host machine	Medium	Unsanitized metadata from the binary package is used by diffoscope and inserted into a shell command. This can lead to remote code execution and consequently, host compromise
Integrity	High	An HTML report can be modified so that the analyzed package does not display malicious changes, which would normally show as a difference in diffoscope	Medium	The malicious package can inject custom HTML into the report. This allows it to manipulate the report's structure, potentially hiding HTML elements that indicate differences in the compared files
Availability	Low	A malicious package crashes strip-nondeterminism, causing the package build process to fail and preventing the package from being released	Easy	An unhandled error occurs when a tool dependency, such as `objdump`, is unavailable, leading to a crash in strip-nondeterminism

Table 5: Risk overview. The threats for Reproducible Builds were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

4.2 Security design coverage check.

Next, the auditing team reviewed the Reproducible Builds design for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.3 Implementation check

As a third step, we tested the current Reproducible Builds implementation for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Reproducible Builds codebase, we derived our code review strategy based on the threat model that we established in the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.1.

Prioritizing by risk, the code was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example, the comparator for ELF binaries in diffoscope
2. Identified viable strategies for the code review. We performed mainly manual code audits, with some static analysis where appropriate
3. Checked that the code did not contain vulnerabilities that could be used to execute the respective attacks. Otherwise, we assessed that sufficient protection measures against specific attacks were present
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations

We carried out a hybrid strategy utilizing a combination of code review and static tests to assess the security of the in-scope tools from the Reproducible Builds codebase.

While static testing establishes a baseline assurance, the focus of this audit was on manual code review of the Reproducible Builds codebases to identify logic bugs, design flaws, and best practice deviations. We reviewed the Reproducible Builds tools repositories up to the revisions shown in table 6. Since the Reproducible Builds codebases is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

Tool	Commit hash	Commit date
diffoscope	0682af7c33b7ff4b120a733da5d75e4d5de45b13	2024-12-06
reprotest	a7fb572f0854cbbedecb1a3c9373d33673b584c6	2024-09-02
strip-nondeterminism	b8e5d87eeb5dabc83ac45b4c6a4923543736c5ea	2024-05-24

Table 6: Audited revisions of in-scope tools

4.4 Remediation support

The final step is supporting Reproducible Builds with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the fix is verified by the auditors to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via the GitLab repositories [1, 2, 3]. Additional communication was mainly done via email with some virtual meetings for clarification of selected questions.

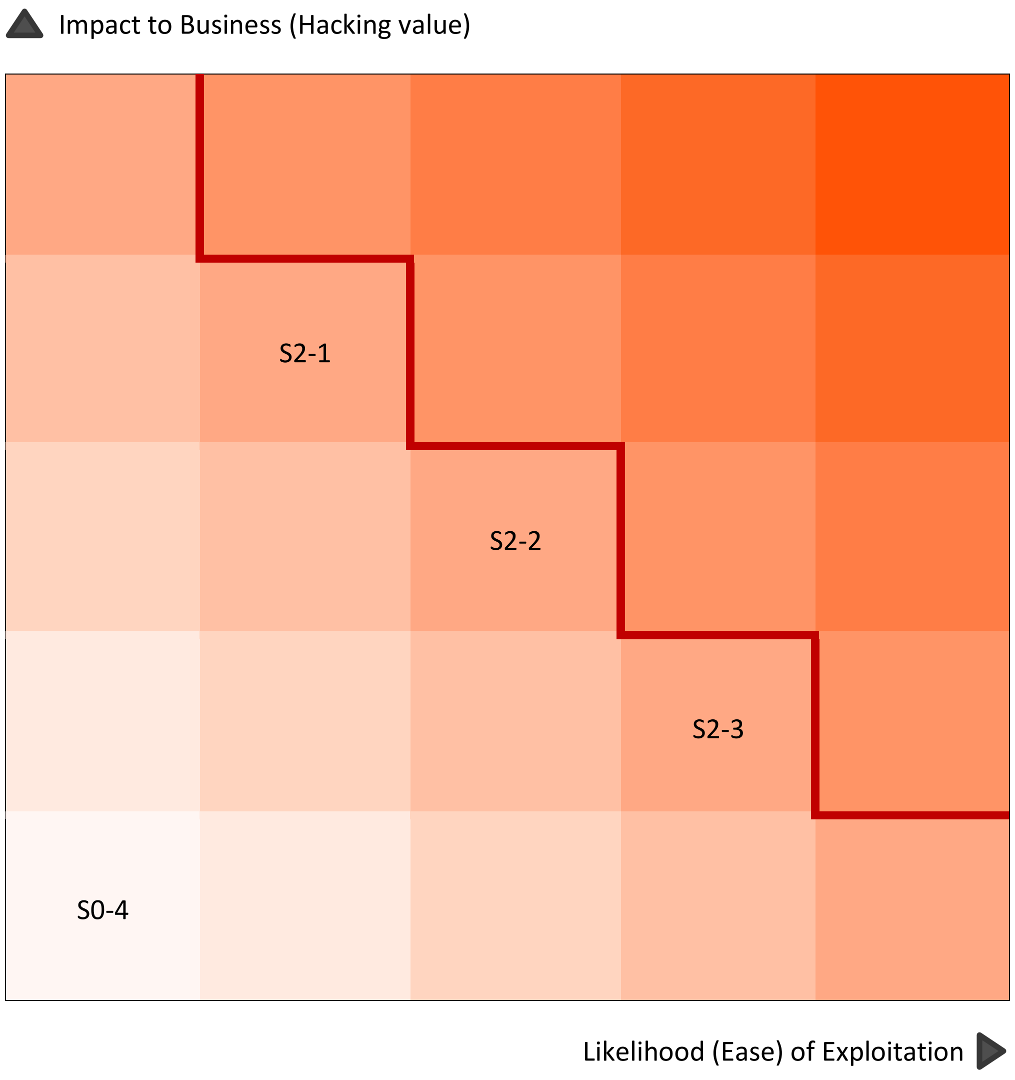
5 Findings summary

We identified 4 issues during our analysis of the Reproducible Builds tools in scope which included diffoscope, reprotest and strip-nondeterminism. In summary, we identified 3 medium-severity and 1 information-level issues. An overview of all findings can be found in table 7.

High	0	
Medium	3	🟧🟧🟧
Low	0	
Informational	1	🟨
Total Issues	**4**	

5.1 Risk profile

The chart below summarizes vulnerabilities according to business impact and likelihood of exploitation, increasing to the top right. The red border separates high and critical security issues from informational to medium ones.



5.2 Issue summary

ID	Issue	Severity	Status
S2-1 [6]	S2-1 Undetected modification in ELF binary	Medium	Acknowledged
S2-2 [7]	S2-2 XML parsing via old versions of Pythons xml.minidom is vulnerable to XEE attacks	Medium	Mitigated [8]
S2-3 [9]	S2-3 Calls to subprocess.check can lead to crashes of diffoscope	Medium	Acknowledged
S0-4 [10]	S0-4 CSS argument is vulnerable to XSS injection	Info	Mitigated [11]

Table 7: Findings overview

6 Detailed findings

6.1 S2-1 Undetected modification in ELF binary

Attack scenario	**An attacker modifies an ELF section without it being detected by diffoscope**
Tool	diffoscope
Tracking	https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/399
Attack impact	Attackers can hide backdoors or altered execution paths inside an ELF binary
Severity	Medium
Status	Acknowledged

Issue description
In diffoscope, when comparing ELF binaries, certain sections, such as `.debug_hello`, are skipped during the comparison process. This is because all sections that start with ".debug" or ".zdebug" and do not end with "_str" are skipped, as checked in the `_should_skip_section` function within diffoscope/comparators/elf.py. Normally, if a difference is detected in a section, the command `readelf --wide --decompress --hex-dump=.custom {}` is invoked to compare the contents. However, for skipped sections, this command is never called.

Although skipped sections are not directly compared, the command `strings --all --bytes=8 {}` would typically still catch the differences. However, this only works for strings longer than eight bytes. If a string shorter than eight bytes is replaced, the change goes undetected by `strings`.

To avoid altering the binary's byte size and triggering metadata changes, an attacker could replace a string with another string of the same size. This allows data in skipped sections like `.debug_hello` to be replaced without detection. An example of this exploit scenario is attached.

Fortunately, if diffoscope cannot detect a difference using ELF tools, it falls back to binary comparison, which reveals the difference. However, in real-world scenarios where developers or maintainers compare an older version of a binary with a new release, other changes detected by ELF tools could mask the undetected modification. If at least one legitimate change is detected, the data replacement in skipped sections could remain unnoticed.

Risk
Undetected modifications in ELF binaries can alter a program's execution flow, leading to malicious or unexpected behavior. In the worst case, this vulnerability could potentially be exploited to hide backdoors within the binaries.

Mitigation
To address this issue, consider displaying the full hex difference either consistently at the end of the comparison or specifically in edge cases. For example, if diffoscope detects that a section was skipped and no changes are found using the `strings` command, it should automatically trigger a binary comparison and display the difference, even when other ELF differences are detected. This approach could help ensure that hidden modifications in skipped sections are properly detected and flagged.

6.2 S2-2 XML parsing via old versions of Pythons xml.minidom is vulnerable to XEE attacks

Attack scenario	**An attacker crafts a malicious XML file that included exponential entities to slow down and eventually crash diffoscope during diff calculation.**
Tool	diffoscope
Tracking	https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/397
Attack impact	Attackers may crash diffoscope to make build pipelines fail and delaying publication of package updates to distributions.
Severity	Medium
Status	Mitigated [8]

Issue description
diffoscope uses `xml.dom.minidom` from Pythons standard library to parse XML DOM content if the safer `defusedxml` package is not installed. The standard library module is vulnerable to two kinds of DOS attack vectors via entity expansion and/or large tokens if the version of the underlying C library `expat` is not recent enough.

As described in Python module documentation [12] versions of `expat <2.4.1` (released on 2021-05-23) are vulnerable to exponential/quadratic entity expansion while versions `<2.6.0` (released on 2024-02-06) are vulnerable to large tokens. Since `expat` is usually provided by the system and not directly bundled with Python, even recent Python installations can still be vulnerable due to old `expat` versions.

Risk
For affected versions, the following example XML file leads to excessive parsing times with high memory usage:

```
<!DOCTYPE xmlbomb [
<!ENTITY a "1234567890">
<!ENTITY b "&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;">
<!ENTITY c "&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;">
<!ENTITY d "&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;">
<!ENTITY e "&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;&d;">
<!ENTITY f "&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;">
<!ENTITY g "&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;&f;">
<!ENTITY h "&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;&g;">
<!ENTITY i "&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;&h;">
]>
<bomb>&i;</bomb>
```

Figure 1: XML bomb proof of concept

On resource-constrained systems this can lead to execution timeouts or even crashes, thus enabling a DOS vector on diffoscope.

Mitigation
The issue is fixed by using recent versions of `expat`. However, as outlined above, this cannot be enforced from diffoscope directly (besides vendoring `expat` directly) due to the reliance on system libraries.

Nonetheless, diffoscope should detect if a vulnerable version is in use by checking `pyexpat.EXPAT_VERSION` and either (1) aborting safely, or (2) continuing with a warning message to inform the user of potential risks. In both cases the paths to mitigation by either installing `defusedxml` (preferred) or upgrading `expat` should be printed to the user.

6.3 S2-3 Calls to subprocess.check can lead to crashes of diffoscope

Attack scenario	**An attacker makes a utility that diffoscope calls return non-zero exit codes such that diffoscope crashes unexpectedly.**
Tool	diffoscope
Tracking	https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/398
Attack impact	Attackers may crash diffoscope to make build pipelines fail and delaying publication of package updates to distributions.
Severity	Medium
Status	Acknowledged

Issue description

diffoscope uses various functions of the python `subprocess` module to call external programs like `xz` or `apktool` to compare complex file formats and archives. The `check_` variants of these functions throw a `CalledProcessError` exception if the called subprocess exits with a non-zero returncode (i.e. the command failed). These function calls are largely placed outside of `try/except` blocks and thus lead to crashes of diffoscope if the subprocess exits unsuccessfully. The following snippets shows an example location where the issue occurs:

```
def open_archive(self):
    [...]

    subprocess.check_call(
        (
            "apktool",
            "d",
            "-f",
            "-k",
            "-m",
            "-o",
            self._tmpdir.name,
            self.source.path,
        ),
        stderr=None,
        stdout=subprocess.PIPE,
    )

    [...]
```

Figure 2: ApkContainer.open_archive (line 72ff.)

This issue also applies to `Command.our_check_output` which essentially just wraps `subprocess.check_output` with an additional logging call. The following snippet shows an instance of the dangerous usage of `our_check_output`:

```
def __init__(self, *args, **kwargs):
    super().__init__(*args, **kwargs)
    [...]

    output = our_check_output(cmd, shell=False, stderr=subprocess.DEVNULL)

    [...]
```

Figure 3: ElfContainer.__init__ (line 457ff.)

While the above sections are examples, all calls to `subprocess` functions that throw exceptions such as `CalledProcessErrors` are problematic and should be mitigated.

Risk
The ease of crafting malicious files depends on the specific external tool being used. Malicious files can cause external tools to exit abnormally, effectively crashing diffoscope. Alternatively, simple forms of argument injection (e.g. from file metadata) can lead to unexpected return codes and thus crashes.

Mitigation
To ensure diffoscope operates correctly, it is essential to verify the return codes of external tools, as their proper execution is critical to its functionality. Additionally, calls should be encapsulated within `try/except` blocks to handle errors gracefully and ensure diffoscope exits safely in case of failures.

This mitigation is already in place in selected locations, sometimes even in the same file as the problematic sections:

```
def get_debug_link(path):
    try:
        output = our_check_output(
            [get_tool_name("readelf"), "--string-dump=.gnu_debuglink", path],
            stderr=subprocess.DEVNULL,
        )
    except subprocess.CalledProcessError as e:
        logger.debug("Unable to get Build Id for %s: %s", path, e)
        return None

    m = re.search(
        r"^\s+\[\s+0\]\s+(\S+)$",
        output.decode("utf-8", errors="replace"),
        flags=re.MULTILINE,
    )
    if not m:
        return None

    return m.group(1)
```

Figure 4: Code location diffoscope/comparators/elf.py (line 416ff)

6.4 S0-4 CSS argument is vulnerable to XSS injection

Attack scenario	**An attacker injects JavaScript in the generated HTML report which executes code leading to XSS**
Tool	diffoscope
Tracking	https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/396
Attack impact	Attackers gets code execution once the report is opened but need to manipulate the arguments of diffoscope to achieve it
Severity	Info
Status	Mitigated [11]

Issue description
Diffoscope allows custom CSS to be loaded for an HTML report. While the HTML input appears to be properly escaped, the `--css` argument, which expects a URL, can be abused to insert custom Javascript instead. For example, the following payload: `--css "\"><svg/onload=alert(43433)>"`, results in an an alert when opening the HTML report.

Risk
An attacker can inject malicious scripts into a trusted differential report, which comes with a variety of problems associated with XSS e.g. reading sensitive data. Although the `--css` argument is typically under user control, this assumption might not hold for all diffoscope use cases. For instance, a web service like `try.diffoscope.org` [13] could, in the future, permit users to specify custom CSS, exposing the service to potential XSS attacks.

Mitigation
Ensure that inputs provided to the `--css` argument are validated as valid URLs, as specified in the `--help` documentation for `--css`.

7 Bibliography

[1] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope.

[2] [Online]. Available: https://salsa.debian.org/reproducible-builds/strip-nondeterminism.

[3] [Online]. Available: https://salsa.debian.org/reproducible-builds/reprotest.

[4] [Online]. Available: https://pypi.org/project/bandit/.

[5] [Online]. Available: https://pypi.org/project/safety/.

[6] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/399.

[7] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/397.

[8] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/commit/889597c91f19dc34d8a4ccc6db213c2ca15d4a21.

[9] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/398.

[10] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/issues/396.

[11] [Online]. Available: https://salsa.debian.org/reproducible-builds/diffoscope/-/commit/a36ee4ebd7494d6d24d537072974a4ae92437523.

[12] [Online]. Available: https://docs.python.org/3/library/xml.html#xml-vulnerabilities.

[13] [Online]. Available: https://try.diffoscope.org/.

Appendix A: Technical services

Security Research Labs delivers extensive technical expertise to meet your security needs. Our comprehensive services include software and hardware evaluation, penetration testing, red team testing, incident response, and reverse engineering. We aim to equip your organization with the security knowledge essential for achieving your objectives.

SOFTWARE EVALUATION We provide assessments of application, system, and mobile code, drawing on our employees' decades of experience in developing and securing a wide variety of applications. Our work includes design and architecture reviews, data flow and threat modelling, and code analysis with targeted fuzzing to find exploitable issues.

BLOCKCHAIN SECURITY ASSESSMENTS We offer specialized security assessments for blockchain technologies, focusing on the unique challenges posed by decentralized systems. Our services include smart contract audits, consensus mechanism evaluations, and vulnerability assessments specific to blockchain infrastructure. Leveraging our deep understanding of blockchain technology, we ensure your decentralized applications and networks are secure and robust.

POLKADOT ECOSYSTEM SECURITY We provide comprehensive security services tailored to the Polkadot ecosystem, including parachains, relay chains, and cross-chain communication protocols. Our expertise covers runtime misconfiguration detection, benchmarking validation, cryptographic implementation reviews, and XCM exploitation prevention. Our goal is to help you maintain a secure and resilient Polkadot environment, safeguarding your network against potential threats.

TELCO SECURITY We deliver specialized security assessments for telecommunications networks, addressing the unique challenges of securing large-scale and critical communication infrastructures. Our services encompass vulnerability assessments, secure network architecture reviews, and protocol analysis. With a deep understanding of telco environments, we ensure robust protection against cyberthreats, helping maintain the integrity and availability of your telecommunications services.

DEVICE TESTING Our comprehensive device testing services cover a wide range of hardware, from IoT devices and embedded systems to consumer electronics and industrial controls. We perform rigorous security evaluations, including firmware analysis, penetration testing, and hardware-level assessments, to identify vulnerabilities and ensure your devices meet the highest security standards. Our goal is to safeguard your hardware against potential attacks and operational failures.

CODE AUDITING We provide in-depth code auditing services to identify and mitigate security vulnerabilities within your software. Our approach includes thorough manual reviews, automated static analysis, and targeted fuzzing to uncover critical issues such as logic flaws, insecure coding practices, and exploitable vulnerabilities. By leveraging our expertise in secure software development, we help you enhance the security and reliability of your codebase, ensuring robust protection against potential threats.

PENETRATION & RED TEAM TESTING We perform high-end penetration tests that mimic the work of sophisticated adversaries. We follow a formal penetration testing methodology that emphasizes repeatable, actionable results that give your team a sense of the overall security posture of your organization.

SOURCE CODE-ASSISTED SECURITY EVALUATIONS We conduct security evaluations and penetration tests based on our code-assisted methodology that lets us find deeper vulnerabilities, logic flaws, and fuzzing targets than a black-box test would reveal. This gives your team a stronger assurance that the significant security-impacting flaws have been found and corrected.

SECURITY DEVELOPMENT LIFECYCLE CONSULTING We guide organizations through the Security Development Lifecycle to integrate security at every phase of software development. Our services include secure coding training, threat moelling, security design reviews, and automated security testing implementation. By embedding security practices into your development processes, we help you proactively identify and mitigate vulnerabilities, ensuring robust and secure software delivery from inception to deployment.

REVERSE ENGINEERING We assist clients with reverse engineering efforts that are not associated with malware or incident response. We also provide expertise in investigations and litigation by acting as experts in cases of suspected intellectual property theft.

HARDWARE EVALUATION We evaluate new hardware devices ranging from novel microprocessor designs, embedded systems, mobile devices, and consumer-facing end products to core networking equipment that powers Internet backbones.

VULNERABILITY PRIORITIZATION We streamline vulnerability information processing by consolidating data from compliance checks, audit findings, penetration tests, and red team insights. Our prioritization and automation strategies ensure that the most critical vulnerabilities are addressed promptly, enhancing your organization's security posture. By systematically categorizing and prioritizing risks, we help you focus on the most impactful threats, ensuring efficient and effective remediation efforts.

SECURITY MATURITY REVIEW We conduct comprehensive security maturity reviews to evaluate your organization's current security practices and identify areas for improvement. Our assessments cover a wide range of criteria, including policy development, risk management, incident response, and security awareness. By benchmarking against industry standards and best practices, we provide actionable insights and recommendations to enhance your overall security posture and guide your organization toward achieving higher levels of security maturity.

SECURITY TEAM INCUBATION We provide comprehensive support for building security teams for new, large-scale IT ventures. From Day 1, our ramp-up program offers essential security advisory and assurance, helping you establish a robust security foundation. With our proven track record in securing billion-dollar investments and launching secure telco networks globally, we ensure your new enterprise is protected against cyberthreats from the start.

HACKING INCIDENT SUPPORT We offer immediate and comprehensive support in the event of a hacking incident, providing expert analysis, containment, and remediation. Our services include detailed forensics, malware analysis, and root cause determination, along with actionable recommendations to prevent future incidents. With our rapid response and deep expertise, we help you mitigate damage, recover swiftly, and strengthen your defenses against potential threats.