Pacman

Security Assessment and Lightweight Threat Model

March 7, 2024

Prepared for:
Levente Polyak
Organized by the Open Technology Fund

Prepared by: **Spencer Michaels, David Pokora, Sam Alws, and Dominik Czarnota**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project managers were associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering directors were associated with this project:

Anders Helsing, Engineering Director, Application Security
anders.helsing@trailofbits.com

The following consultants were associated with this project:

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
November 13, 2023	Pre-project kickoff call
November 14, 2023	Discovery meeting #1
November 15, 2023	Discovery meeting #2
November 16, 2023	Discovery meeting #3
December 5, 2023	Delivery of report draft, threat model readout meeting
December 6, 2023	Code review readout meeting
March 7, 2024	Delivery of fix review appendix

Executive Summary

Engagement Overview

The Open Technology Foundation engaged Trail of Bits to review the security of the Pacman package manager, as well as its closely-associated package management library `libalpm`. Pacman is the official package manager of Arch Linux and is developed by the Arch team; it is also used in a handful of other Linux distributions, including Manjaro.

A team of two consultants conducted a threat model from November 13th to 17th, for a total of two engineer weeks; this was followed by a code review by three engineers from November 20th to December 1st, for a total of five engineer-weeks of effort. Our testing efforts focused on package signature verification, data integrity during downloads and upgrades, memory safety, and a new user-based isolation mechanism. With full access to source code and documentation, we performed static and dynamic testing of Pacman and `libalpm`, including fuzzing, using automated and manual processes. The audit scope excluded the parts of the Pacman ecosystem used exclusively for building packages, such as `makepkg`.

Observations and Impact

Overall, Pacman is well-designed, comprehensively-documented, and robust against common application security issues. The code review portion of the engagement revealed several issues ranging from low to undetermined severity, and while the threat model revealed some plausible threat scenarios, these generally require the confluence of several independent factors which set a relatively high bar for an attacker to achieve, such as compromising a mirror, obtaining a signing key, intercepting a user's connection under certain configurations, and so on.

That said, certain defense-in-depth measures can be implemented to improve the resilience of Pacman and the Arch Linux distribution and signing infrastructure, even against cases where an attacker already has a partial foothold. Based on the threat model and code review results, three major areas of improvement stand out:

- As Pacman is written in C, even security-conscious developers run a relatively high risk of accidentally introducing memory safety issues — we discovered several during the audit, although ultimately none proved especially serious (TOB-PACMAN-1, TOB-PACMAN-4, TOB-PACMAN-9). We recommend employing the use of static and dynamic analyses, including fuzz tests, to uncover additional potential cases of memory corruption and leaks before attackers do.
- Pacman's signing infrastructure is robust against maintenance issues such as keys being lost (not stolen), signers becoming inactive or incapacitated, and so on. However, due to a lack of documented incident response procedures, the Arch Linux team may be ill-equipped to promptly respond to a security incident involving

theft or malicious use of key materials. Additionally, a lack of clear auditing guidelines and trust requirements for signers increases the likelihood that package signing keys could be used maliciously. As Arch Linux continues to grow as an organization, it is critical that security-related processes, guidelines, and requirements are clearly and precisely documented to ensure consistency and prompt response to security incidents.

- Pacman can verify database signatures, but Arch Linux's official databases are not signed and Pacman does not require databases to be signed by default. Combined with the fact that Pacman allows the use of plaintext HTTP package mirrors, users with such a configuration could be served malicious database files, which could serve old and vulnerable versions of packages. This issue is known to the Arch Linux team, and work is currently underway to rectify it.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Arch Linux team take the following steps:

- **Remediate the code review findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Create a long-term plan for implementing the strategic recommendations in the Threat Model section of this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Clarify the intended use and safety guarantees of the `--root` argument.** This argument specifies which directory should be used by Pacman as the root directory. However, it is not guaranteed that files and directories *outside* of the root directory will remain untouched (for example, if there is a maliciously placed symlink inside of the root directory). Pacman's manpage entry states that the argument should not be used as "a way to install software into `/usr/local` instead of `/usr`" or "for performing operations on a mounted guest system". However, Pacman documentation does not state what this argument *should* be used for, and does not give any information about the argument's (lack of) safety guarantees.

- **Implement security-focused static analysis, dynamic analysis, and fuzz tests.** These should be run against each new Pacman version prior to release to minimize the likelihood that ongoing code changes introduce memory corruption issues.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	1
Informational	5
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Denial of Service	1
Undefined Behavior	3

Project Goals

The engagement was scoped to provide a security assessment of the Pacman package manager. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is there any way to bypass Pacman's package signature validation?
- Is it possible to break out of the SandboxUser's filesystem context implemented in MR 23?
- Does the package consistency checking included in MR 96 have any security issues?
- Is Pacman vulnerable to any form of memory corruption?
- Can an attacker with control over database contents (which are unsigned by default and may be accessed over plaintext HTTP) cause Pacman to exhibit malicious behavior?
    - In particular, can a malicious database silently downgrade a package to a known-vulnerable version, install a vulnerable package, or uninstall a package providing security measures?
- Can a malformed package, or malformed metadata, cause Pacman to bring the system into an inconsistent state?
- Are Pacman's defaults conducive to secure operation by ordinary users?
- Does Pacman call out to third-party programs or libraries in unsafe ways?
- Does Pacman's current test suite appropriately cover security related concerns?
- Is Arch Linux's package signing infrastructure robust against failures and resilient to compromise, including malicious insiders?
- Are Arch Linux's official package repositories reasonably well protected against the unexpected introduction of malicious code or metadata?

Project Targets

The engagement involved a review and testing of the target listed below, including two as-yet-unmerged pull requests.

Pacman

Repository	https://gitlab.archlinux.org/pacman/pacman/
Version	`18e49f2c97f0e33a645f364ed9de8e3da6c36d41`
Type	C binary application
Platform	Linux

Merge Request 23: Add SandboxUserConfiguration

URL	https://gitlab.archlinux.org/pacman/pacman/-/merge_requests/23

Merge Request 96: Check package consistency when installing

URL	https://gitlab.archlinux.org/pacman/pacman/-/merge_requests/96

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- A lightweight threat model of Pacman and the portion of its infrastructure related to package signing and distribution.
- Non-exhaustive manual review of the Pacman codebase as well as two security-relevant pull requests pending acceptance, with a focus on code paths pertaining to security-critical functionality highlighted in the initial threat model
- Static analysis of the Pacman codebase and manual triage of results
- Dynamic analysis to identify instances of memory corruption and leaks
- Fuzzing to identify inputs that could cause unexpected behavior at runtime

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Code of various dependencies used by Pacman like libarchive, gpgme etc.
- Although we included signing/packaging infrastructure security controls in the threat model, we did not have access to review their implementation during the code review.

Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model, drawing from Mozilla's "Rapid Risk Assessment" methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). The results of the lightweight threat model are noted in the subsections below.

Data Types

The target application makes use of the following data formats:

- Tar files (.tar), usually compressed (.zst, .gz, or .xz): Pacman package files
- Bash scripts: PKGBUILD, INSTALL files
- INI configuration files: hooks, configuration files (e.g. pacman.conf)
- Plain text: PKGINFO, BUILDINFO, database and file-list files
- PGP keys

Data Flow

Pacman is the default package manager for Arch Linux, maintained officially by the Arch Linux development team.

Pacman retrieves packages from one or more *repositories*, which can either be located on the local host's filesystem, or accessed over the network via any protocol supported by libcurl, which Pacman uses internally. Packages can also be directly installed from the filesystem without being associated with a repository.

In a typical use case, users download the vast majority of their packages pre-built from HTTP or HTTPS mirrors of the remote Arch Linux official repositories. A small number of unofficial packages, such as those from the Arch User Repository, may be built and installed either directly as a manually-built package file on the local filesystem, or from a repository hosted on the local filesystem.

When a package is installed, Pacman verifies its signature using an internal *Pacman Keyring*, with root-of-trust derived from a unique *System Master Key* which is generated upon system installation, and used to sign the set of *Main Signing Keys* imported into the system-local Pacman keyring. These *Main Signing Keys*, of which there are only a small number, are used by the Arch Linux developers to sign *Packaging Keys*, with which package maintainers sign their packages. Each main signing key has an associated *Revocation Key*, held in the possession of a different trusted signer, which can be used to revoke it in the event of a compromise. Those keys, along with the names of developers they belong to, are listed on the https://archlinux.org/master-keys/ website.

Data for keys stored in Pacman's internal keyring is kept-up-to-date via the *WKD Sync Service*, which runs weekly on Arch Linux and syncs with a distributed *Web Key Directory*. The sync service can only update metadata (such as expiration dates) for existing stored keys; it cannot alter whether or not a given key is trusted.

Maintainers generally use *makepkg* to generate pacman packages from application/library sources. The build scripts for Arch Linux's official packages are hosted on a dedicated *GitLab* account, https://gitlab.archlinux.org/, with login handled by an Arch-managed *Keycloak SSO* instance. Most official packages are built on a single, high-capacity *Main Build Server* administered by dedicated *DevOps* members of the Arch Linux team and accessible via SSH; however, individual maintainers may build and sign packages on other machines. Packagers are currently strongly encouraged, although not strictly required, to retain signing keys only on hardware keys (as opposed to on their local filesystem).

All official Arch Linux packages are currently signed, and by default, Pacman requires packages from remote repositories to have a valid signature trusted by Arch's main signing keys. Package installation transactions may be preceded and/or followed by *hooks*, which can invoke arbitrary commands in response to the presence of specific packages in a transaction. Packages, along with detached signatures, are cached in a *Package Cache* directory (`/var/cache/pacman/pkg`) on the local filesystem after installation.

Below, we depict known connections between system components of the package-consumption side of Pacman, as integrated in Arch Linux. These diagrams are intended to convey our understanding of the system as a whole. Further details will be discussed in the Components and Trust Zones and Trust Zone Connections report subsections. The dotted lines indicate trust boundaries separating zones.

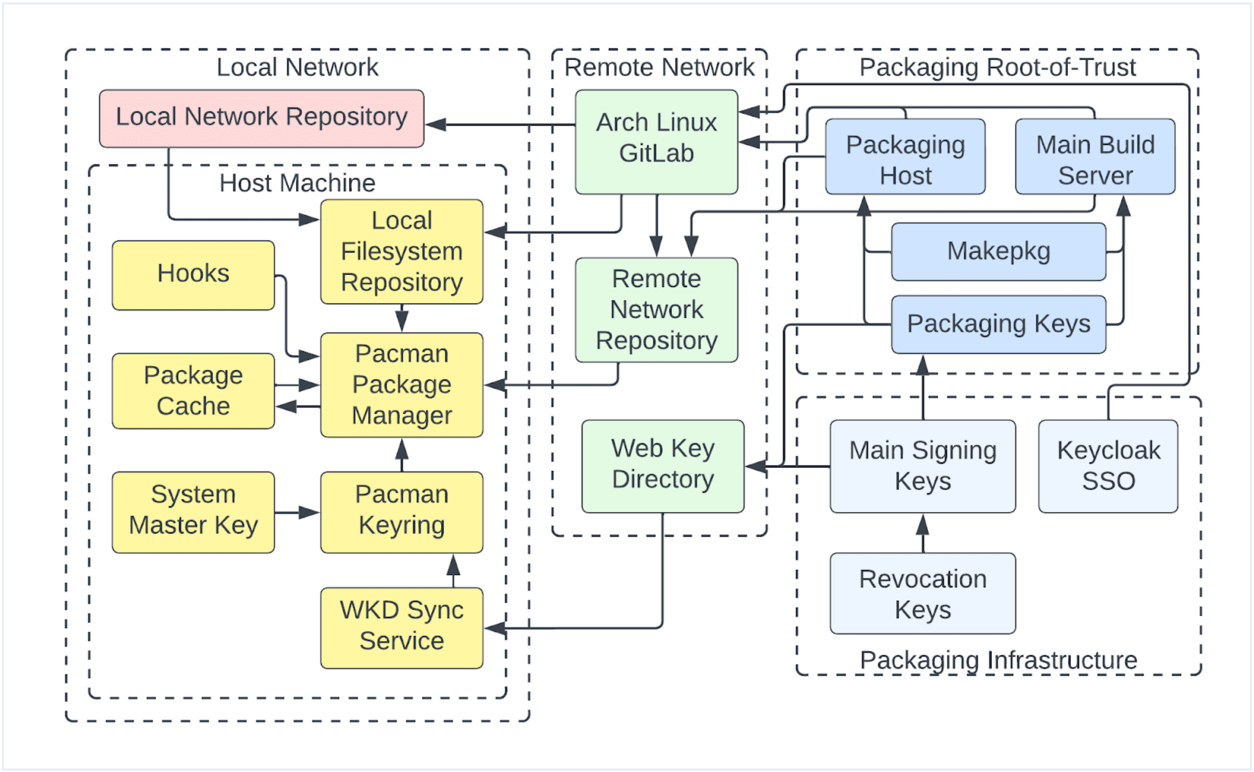

Figure 1: The data flow of packages and their signing data from Arch Linux's root-of-trust to the host machine on which Pacman runs.

Components and Trust Zones

The following table describes the components that make up the Pacman package management system, as well as the external dependencies on which they rely. These system elements are further classified into *trust zones*—logical clusters of shared functionality and criticality, between which the system enforces (or should enforce) interstitial controls and access policies.

Components marked by asterisks (*) are considered out of scope for the assessment. We explored the implications of threats involving out-of-scope components that directly affect in-scope components, but we did not consider threats to out-of-scope components themselves.

Component	Description
Host Machine	The host on which Pacman is used to manage packages.
Pacman Package Manager	Pacman is a package management tool that tracks installed packages on a Linux system, including support for dependency resolution/retrieval, package groups, install/uninstall scripts, and pre/post-install hooks. It also

	contains utilities such as makepkg, used to create packages which can be installed by Pacman.
Local Filesystem Repository	A repository residing on the local filesystem. Repositories provide a listing of packages which can be fetched, installed, or upgraded. The package listing is managed by the repository maintainer(s). Packages can be signed or unsigned.
Package Cache	A directory populated with previously-installed packages. Cached packages are used to rapidly reinstall a previously installed package. Any signatures contained within packages are also stored alongside and validated for each cached package item.
Pacman Keyring	A keyring containing signing keys for all packages installed on the system. Keys within the keyring are only considered trusted if they are signed by an Arch Linux packaging key (which is in turn signed by a main signing key).
System Master Key	The root of trust for Pacman's signature validation on any given installation. Master keys are generated at first Pacman run (and so on first boot of Arch Linux), are unique to each Arch Linux install, and are used by the host machine to trust the Arch Linux main signing keys.
WKD Sync Service	A GPG wrapper service on Arch Linux that runs weekly to sync updates (e.g. expiry extensions) to keys in the Pacman keyring, pulled from a Web Key Directory (WKD). The WKD service can add previously-unknown signatures to the keyring, but cannot make Pacman trust those signatures.
Hooks	Pre- and post-install hooks which enable running commands just before or after a Pacman transaction (e.g., to rebuild a new kernel image after Pacman installs a new kernel version).
Local Network	The components which share a local network with the Host Machine.
Local Network Repository	A repository residing. Repositories provide a listing of packages which can be fetched, installed, or upgraded. The package listing is managed by the repository maintainer(s). Packages can be signed or unsigned.
Remote Network	The components which live outside of the Local Network trust zone, e.g. public-facing external network components.
Remote Network Repository	A repository residing on a remote network host. Repositories provide a listing of packages which can be fetched, installed, or upgraded. The

	package listing is managed by the repository maintainer(s). Packages can be signed or unsigned.
Web Key Directory (WKD)	GnuPG's standard system for key discovery, which maps public keys to email addresses.
Arch Linux GitLab (*)	The GitLab account hosting the source code for official Pacman packages.
Packaging Infrastructure	The machines (and their operators) that build and sign Pacman packages.
Main Build Server (*)	The dedicated, high-capacity machine that the Arch Linux team uses to build the majority of its official packages.
Packager Host	A host operated by a Packager, used to build and sign packages.
Packaging Keys	A key used by package maintainers to sign packages. Each trusted maintainer is issued a packaging key signed by a quorum of main signing keys.
Makepkg (*)	The toolset used to build Pacman packages.
Packaging Root-of-Trust	The components which are used to facilitate administration of an operating system's primary mirrors and managing their authorized package signers.
Main Signing Keys	The root of trust for Arch Linux's signing infrastructure, which can sign new packaging keys as well as packages themselves. Currently, only five main signing keys exist.
Revocation Keys	Each Main Signing Key has a single associated Revocation Key used to revoke the signing key in the event of compromise. Each signing key's Revocation Key is held by another signing key owner.
Arch Linux DevOps	The individuals who administer Arch Linux's Keycloak SSO instance, GitLab account, etc.

Trust Zone Connections

At a design level, trust zones are delineated by the security controls that enforce the differing levels of trust within each zone. Therefore, it is necessary to ensure that data cannot move between trust zones without first satisfying the intended trust requirements of its destination. We enumerate such connections between trust zones below.

Originating Zone	Destination Zone	Data Description	Connection Type	Auth Type
Host Machine	Host Machine	All operations performed by pacman which leverage components in the same zone, largely rely on cryptographic verification (e.g. signed packages, packager key authorization). The artifacts written by pacman are done so in root-user execution context, with file permissions blocking	Filesystem	File Privileges, GNUpg signature validation
Remote Network	Host Machine	The host's WKD Sync Service pulls updated key information from a Web Key Directory into the Pacman Keyring.	HTTPS	None
Remote Network, Local Network	Host Machine	The host installs a package from a Local or Remote Network Repository.	libcurl supported protocols (e.g. HTTP, HTTPS, FTP, ...)	GNUpg signature validation, libcurl supported protocols (e.g. TLS)
Remote Network	Local Network	Third-party package sources pulled from the Internet are	Varies; likely HTTP/S	Varies or None

		downloaded to, and built on, the local network; the resulting packages are placed in a Local Network Repository.		
Remote Network	Packaging Infrastructure	Package sources hosted on the Arch Linux GitLab are downloaded to, and built on, the Main Build Server or a Packager Host.	HTTPS	SSH
Packaging Root-of-Trust	Packaging Infrastructure	A quorum of Main Signing Key holders signs a new Packaging Key, or issues revocations for an existing one.	N/A	GNUpg signature verification
Packaging Root-of-Trust	Arch Linux GitLab	An Arch Linux administrator logs into GitLab through Keycloak SSO.	HTTPS	OAuth

Threat Actors

When conducting a threat model, we define the types of actors that could threaten the security of the system. We also define other users of the system who may be impacted by, or induced to undertake, an attack. For example, in a confused deputy attack such as cross-site request forgery, a normal user who is induced by a third party to take a malicious action against the system would be both the victim and the direct attacker. Establishing the types of actors that could threaten the system is useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities. We will refer to these actors in descriptions of the security findings that we uncovered through the threat modeling exercise.

Actor	Description
End Users	Actors representing users of Pacman and consumers of its packages and repositories. They operate in the Host Machine zone, and may have influence over the Local Network zone and its repositories.
Local User	A low-privileged user on the Host Machine, e.g. non-admin, non-root. They cannot execute sensitive pacman operations, as they require root-access.
Local Root	The root user on the Host Machine, with privileges to perform any operations they desire. Pacman requires a Local User to elevate to Local Root to install or update packages.
Operators	Privileged actors with the responsibility of operating Packaging Infrastructure and Packaging Root-of-Trust components.
Repository Administrator	An individual with control over a Pacman repository/mirror. They may operate a local machine, local network, or remote repository.
DevOps Administrator	An individual with control over Arch Linux's DevOps infrastructure, including the Arch Linux GitLab account and Keycloak SSO instance.
Packager	An individual in possession of a Packaging Key which was signed and approved by a Trusted Signer
Trusted Signer	An individual in possession of a Master Signing Key, a single keypair used in a threshold signature scheme (TSS) which performs sensitive operations such as approving a new Packaging Key.

	Trusted Signers, in quorum, act as a root of trust for pacman repository management.
Attacker	An attacker positioned either within or external to any of the trust zones previously described.
Internal Attacker	An Internal Attacker is an attacker who has transited one or more trust boundaries. Such an attacker may be an existing actor role in the system or an External Attacker who has successfully transited a trust boundary into the system.
External Attacker	An External Attacker is an attacker who is external to the cluster and is unauthenticated, such as an attacker with control over external services.

Threat Scenarios

The following table describes possible threat scenarios given the design, architecture, and risk profile of the Pacman package manager.

Scenario	Actor(s)	Component(s)
An operating system provides a default mirror list leveraging insecure protocols. Developers of an operating system such as Arch Linux may generate a list of repository sources which leverage insecure protocols (e.g. HTTP, FTP). Due to pacman's lack of protocol restrictions, its underlying libcurl dependency will communicate over the insecure protocol. If a Local User or Local Root actor uses this insecure protocol to fetch packages from a Local Network or Remote Repository, it may expose them to man-in-the-middle attacks. Although such an attack may not be problematic for signed packages, unsigned packages may be substituted with maliciously crafted packages by an Attacker.	• Repository Administrator • Trusted Signer • Attacker	• Pacman Package Manager
An operating system which leverages pacman does not enforce signed packages by default. Arch Linux by default requires all packages to be signed to be installed, verifying they have been approved. In the event a Linux distribution does not configure pacman to require signatures, this may introduce risk, compounding on the threat scenario mentioned in the previous row of this table. Unsigned packages may be modified or indicative of a lack of approval process. They may be subject to modification in-flight through a man-in-the-middle attack that may put users at risk. The Package Cache containing a copy of previously unsigned installations may also be modified if it is improperly secured. By default, Arch Linux saves Package Cache items with special privileges that should disallow any user role except Local Root to modify them, mitigating this risk.	• Repository Administrator • Packager • Trusted Signer • Attacker	• Pacman Package Manager

An environment variable affects Pacman Package Manager's libcurl dependency. For instance, Pacman redirects its HTTP connections through the proxy defined in the `http_proxy` environment variable. If an attacker injects an environment variable into Pacman's runtime environment — a difficult prospect, given that it runs as root during installs — he may be able to cause Pacman to exhibit exploitable or undesirable behavior.	• Local Root	• Pacman Package Manager
An Attacker attempts a substitution attack, bumping versions on a popular package through a compromised Local Network Repository or Remote Repository. Pacman will always install the latest version of a package across all repositories it has access to. As such, if a user has both local and remote repositories enabled, an attacker who is able to introduce an identically-named, higher-versioned package into one of the remote repositories can easily induce the user to install his version of the package. Similar attacks may also be possible via DNS confusion, e.g. if an attacker registers a domain that shadows a local-network domain name. See this GitHub blog post on substitution attacks against NPM.	• Repository Administrator • External Attacker	• Pacman Package Manager • Local Network Repository • Remote Network Repository
An attacker compromises a Packaging Key and produces different, but valid, signatures for a package to introduce malicious changes. In this case, Pacman would install the new package version normally, and the user would be entirely unaware. Currently, there is no way to enable a warning when a package's signature changes.	• Packager • Internal Attacker	• Pacman Package Manager • Packaging Keys
A Packaging Key or Packager is compromised, requiring revocation of their Packaging Key. Due to a lack of documented procedures for revocation, response by Trusted Signers may be delayed, giving the attacker more time to cause damage.	• Packager • Trusted Signer	• Packaging Keys

A Trusted Signer's key is compromised, requiring incident response. Due to a lack of documented procedures for revocation, response by the Trusted Signer holding the compromised key's revocation key may be delayed, giving the attacker more time to cause damage.	• Trusted Signer	• Main Signing Keys • Revocation Keys
The (unsigned) Pacman database used to index packages may be modified by an Attacker. The database used by Pacman is not signed. The database is used as an index for packages on the system. Although most of the data used by Pacman is derived from signed packages on Arch Linux, the database is used to determine depends/replace directives when installing a package. This is done without verification that the depends/replace data taken from package metadata has not been tampered with. As such, an Attacker with access to the Pacman database may replace depends/replace directives within the database for a given package, to trigger a deletion or replace-with-empty operation of an existing package on the user's system.	• End User • Internal Attacker • Packager	• Pacman Package Manager • Local Network Repository • Remote Network Repository
Vulnerable or malicious packages are assigned a package group with a name identical to a popular, existing package. Currently, Pacman always resolves such a conflict in favor of the group, with no way to override this behavior. As such, users could be made to unwittingly install an arbitrary package or set of packages in place of a common package.	• Packager • End User	• Pacman Package Manager • Local Network Repository • Remote Network Repository
A naive user sets overly-permissive file permissions on their keyring, config files, or hooks. An attacker who achieves local filesystem access — e.g., by compromising a low-privileged	• End User	• Host Machine

service — could inject malicious settings, commands, or additional trusted keys in order to perform privilege escalation.		
A revocation certificate or signing key (e.g. Package Key, Trusted Signer key) is lost or corrupted. In addition, since there are no standard procedures for regular checks of keys or their backup media after initial creation, it is possible that keys could be permanently lost. In particular, since revocation keys are long-lived and very rarely used, they may become inaccessible (e.g. through corrupted media) long before this fact is discovered, only to be realized too late when the key is sorely needed.	• Trusted Signer • Packager	• Revocation Keys • Main Signing Keys
An attacker compromises a mirror of Arch Linux official packages, or intercepts a user's non-TLS connection to a repository, and injects a malicious version of a package. In this case, Pacman would refuse to install the package, as it requires signatures from remote repositories by default.	• End User • Repository Administrator • Internal Attacker	• Pacman Package Manager • Local Network Repository • Remote Network Repository

Recommendations

Trail of Bits recommends that the Arch Linux team implement the following recommendations to mitigate the threat scenarios described above:

1. **Set Pacman to reject non-TLS mirrors by default.** Since databases are not currently signed, an attacker who can intercept an unauthenticated connection between a user and a repository could modify their contents in transit. A new configuration value such as "AllowInsecureMirrors" can be added to pacman.conf to permit the use of non-TLS mirrors on a case-by-case basis if necessary for backwards-compatibility.
    - **Consider also allowing users to set a minimum TLS version in pacman.conf**, defaulting to at least TLS 1.2 (disabling specific ciphersuites supported in 1.2 that are known to be weak) or, ideally, TLS 1.3. Otherwise, HTTPS downgrade attacks may be possible against TLS-enabled mirrors that support older, insecure TLS versions.
    - Update the official Pacman mirror lists to exclude non-TLS mirrors, and consider modifying the `reflector` mirror list ranking tool to take TLS settings into account (i.e. ranking mirrors with stricter settings higher).
2. **Transition to signed databases and require them by default.** Currently, Pacman gets packages' depends/replaces lists from the database. With databases being unsigned, an attacker with the ability to modify them could induce a user to install or remove arbitrary packages.
    - A patch is currently in progress that would check package metadata as listed in the database against the metadata contained within the actual signed package to be installed, which partially mitigates this issue.
3. **Warn users (or give them the option to be warned) when a package's signature changes during an upgrade, even if the signature is valid.** This will provide a defense-in-depth measure against cases where an attacker gains possession of a valid signing key and signs a package not previously signed with that key.
    - Consider introducing a setting into pacman.conf that would toggle these warnings between "off", "print only", and "pause upgrade and interactively ask for confirmation to continue" — the latter case being suitable for especially cautious users. Depending on how often packages' signing keys change in legitimate cases, the default of this setting could be either "print only" (if rare) or "off" (if common).
4. **Provide an interactive resolution prompt in cases where a package and a group both exist with the same name.** Currently, Pacman considers group names

to “shadow” identically-named packages; as such, an attacker who can tag a malicious or vulnerable package as belonging to a group with the same name as a common package — for instance, by manipulating an unsigned database — can cause users to unwittingly install the package of his choice. In the event of a conflict, the user should be prompted to make an explicit selection (in the same manner as “provides” induces).

5. **Have Pacman refuse to load its keyring, config files, or hooks if they are writable by users other than root.** Analogous to SSH’s permissions checks on the `~/.ssh` directory, this prevents users from unknowingly directing Pacman (which is likely running as root) to use a keyring, configuration, or hook that a lower-privileged user or service could maliciously modify, which could permit privilege escalation.

6. **Establish a detailed, written incident response plan that defines how to respond to high-severity threat scenarios,** especially the following. The plan should detail precisely who is responsible for threat response, and the exact steps they should follow to mitigate the threat. Having such guidance in place ensures that there is no ambiguity about how to handle a security incident when it actually happens, ensuring the fastest and most thorough response possible.

   ○ Compromise of a Main Signer Key.

   ○ Compromise of a Packager Key.

   ○ Compromise of a DevOps-managed property such as the Arch Linux Gitlab account, Keycloak SSO instance, etc.

7. **Establish procedures for regularly validating Trusted Signers’ Main Signer Key and Revocation Key backups over time**, to ensure that they remain usable and readily accessible. In addition, provide detailed guidance on how operators should configure and use cold storage backups, ensuring redundancy in case their primary keypair is corrupted.

   ○ Test not only the accessibility and integrity of the backup media, but also the viability of the keys in question: for instance, import revocation keys into a test keyring on a regular basis to ensure that they do indeed revoke the expected signing keys.

8. **Establish a written list of procedures and requirements for onboarding a new Trusted Signer.** Currently, any potential new Trusted Signer must be well-known to the Arch Linux team, and a long-term participant within the Arch ecosystem, meaning that candidates are already extensively vetted. Formalizing this process would reduce the likelihood of mistakes or exceptions being made.

    - **Consider verifying trusted signers' legal IDs.** The current onboarding process, while in effect vetting candidates' real-world identities quite extensively, does not require actual verification of their legal IDs. Doing so would add an additional layer of defense-in-depth and better allow the Arch team to hold a defecting signer legally accountable if necessary.

9. **Establish standards for regular check-ups on Packagers.** Currently, Trusted Signers make a best-effort attempt to identify Packagers who are inactive or are not fulfilling their duties; however, this is not done systematically or at regular intervals. To minimize the chance that inactive or irresponsible signers slip through the cracks.

10. **Establish clear security guidelines for** Trusted Signers and Packagers, including how to generate, store, and use key material, how to report a compromise of their own key material, what to do if a Trusted Signer reports a compromise, and so on.

    - **Notably, require Packagers to keep key material on hardware keys only.** Currently, this practice is strongly encouraged, but not mandated, and some Packagers sign using key material on their local filesystems.

11. **Consider replacing uses of MD5 with a hashing algorithm with a lower chance of hash collision, such as BLAKE2.** MD5 has a nontrivial chance of collisions, and it is feasible to intentionally craft a file with a specific MD5 hash. Some .pacsave backups, which use MD5 hashing to compare files, may not occur in the case of a hash collision even if the files in question do actually differ. However, performance or compatibility considerations may prohibit the use of algorithms with lower rates of hash collision.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
scan-build	A static analysis tool that can find various issues within C/C++ codebases.	Default checks
libFuzzer	An in-process, coverage-guided, evolutionary fuzzing engine. LibFuzzer can automatically generate a set of inputs that exercise as many code paths in the program as possible.	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- The program does not access invalid memory addresses.
- The program does not exercise undefined behavior.

Test Results

The results of this focused testing are detailed below.

Fuzzing harnesses. The fuzzing harnesses we developed that exercise a subset of the program's code.

Property	Tool	Result
`fuzz_string_length` – harness that checks one of utility functions that computes the length of a string, omitting ANSI escape codes	libFuzzer	TOB-PACMAN-4
`fuzz_wordsplit` – harness that checks one of utility functions that splits a string into multiple words	libFuzzer	Did not find issues

`fuzz_parseconfigfile` – harness that tests the parsing of config files. Requires further changes so it is chrooted and so that the parser doesn't include external files from the file system.	libFuzzer	**Requires further development (see Appendix D)**
`fuzz_alpm_extract_keyid` – harness that tests the extraction of keys from signature data	libFuzzer	TOB-PACMAN-9

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Although the code attempts to test the computed indexes or path lengths, the project does not take specific measures to ensure arithmetic safety. For example, we found an instance where an integer underflow occurred in a length check function. Additionally, oftentimes the length check values are computed from hardcoded integer constants, instead of using the `sizeof()` operator to compute the length of the hardcoded string from which the integer constant length is derived.	**Moderate**
Auditing	Pacman generally preserves standard error from subprocesses (e.g., hooks), and produces useful, detailed messages when package operations encounter errors.	**Satisfactory**
Authentication / Access Controls	Pacman itself does not require authentication or attempt to authenticate to other services.	**Not Applicable**
Complexity Management	Pacman's codebase is neatly organized, with discrete functionality organized into separate files and functions, accompanied by clear comments and documentation.	**Strong**
Configuration	Pacman calls out to well-vetted third-party libraries for complex functionality such as downloads (libcurl) and signature verification (OpenSSL), and uses those libraries according to their respective best practices.	**Strong**
Cryptography and Key Management	Pacman uses OpenSSL for all cryptographic operations. Arch Linux's signing infrastructure has built-in resilience measures such as physical key backups, quorum requirements, and public oversight. However, no written	**Satisfactory**

Category	Summary	Result
	incident response plans exist. This could increase the team's response time in the event the signing infrastructure is compromised.	
Data Handling	While the code generally attempts to verify the data it receives, there were certain cases where the performed checks were insufficient and could cause memory corruption or undefined behavior.	**Moderate**
Documentation	Pacman and `libalpm` are both extensively documented, including in code comments, documentation, man pages, and on the Arch Linux wiki.	**Strong**
Maintenance	Some issues were discovered in how Arch Linux team members maintain the signing infrastructure itself. While the Arch team occasionally audits package signers on an informal basis, no formal process has been defined for how, and how often, such audits should take place. In addition, revocation key backups are not checked regularly after they are first generated; if a backup fails, signers may be unable to revoke a compromised key in a timely manner.	**Moderate**
Memory Safety and Error Handling	We uncovered some instances of memory safety issues where certain parsing routines were able to read memory out-of-bounds. We recommend fuzzing those and other code paths regularly to cover more edge cases and help catch new problems. Errors are generally handled consistently within the codebase, though there were cases where allocation failures were not acted upon apart from logging, though this could be hard to recover from. Additionally, the code could benefit from better distinction of status code return type for its public functions (instead of being an `int` type).	**Weak**
Testing and Verification	Pacman has substantial test coverage for expected functionality, but none that focuses on unexpected inputs or potentially malicious behavior (e.g. fuzz tests).	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Use-after-free vulnerability in the print_packages function	Undefined Behavior	**Low**
2	Null pointer dereferences	Denial of Service	**Informational**
3	Allocation failures can lead to memory leaks or null pointer dereferences	Undefined Behavior	**Informational**
4	Buffer overflow read in string_length utility function	Data Validation	**Undetermined**
5	Undefined behavior or potential null pointer dereferences by passing null pointers to functions requiring non-null arguments	Data Validation	**Undetermined**
6	Undefined behavior from use of atoi	Undefined Behavior	**Informational**
7	Database parsers fail silently if an option is not recognized	Data Validation	**Informational**
8	Cache cleaning function may delete the wrong files	Data Validation	**Informational**
9	Integer underflow in a length check leading to out-of-bounds read in alpm_extract_keyid	Data Validation	**Undetermined**

Detailed Findings

1. Use-after-free vulnerability in the print_packages function

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-PACMAN-1
Target: `pacman/src/pacman/util.c`	

Description

The `print_packages` function has a use-after-free vulnerability. It first deallocates memory for the `temp` variable and then uses that memory in the `PRINT_FORMAT_STRING` macro (figure 1.1). This can lead to:

- Potential exploitation of the program if an attacker would be able to allocate and control the content of the `temp` variable after it is freed (1) and before it is used (2) in another thread. Note that the time window for it is very small since the two operations happen one after another.
- A double free which if detected by the allocator, would cause a program crash. The second free is called in the `PRINT_FORMAT_STRING` macro.

The severity of this finding is low since the first scenario should not be possible because Pacman doesn't use multiple threads.

This issue has been found with the scan-build static analyzer.

```
void print_packages(const alpm_list_t *packages) {
...
/* %s : size */
if(strstr(temp, "%s")) {
        char *size;
        pm_asprintf(&size, "%jd", (intmax_t)pkg_get_size(pkg));
        string = strreplace(temp, "%s", size);
        free(size);
        free(temp); // (1) memory pointed by the temp variable is freed
}
/* %u : url */
PRINT_FORMAT_STRING(temp, "%u", alpm_pkg_get_url) // (2) use-after-free of temp
```

Figure 1.1: pacman/src/pacman/util.c#L1258-1267

```
#define PRINT_FORMAT_STRING(temp, format, func) \
       if(strstr(temp, format)) { \
              string = strreplace(temp, format, func(pkg)); \
              free(temp); \
              temp = string; \
       } \
```

Figure 1.2: The PRINT_FORMAT_STRING macro definition

This issue can also be detected with tools such as Valgrind (figure 1.3) or AddressSanitizer.

```
# valgrind ./pacman -S --print --print-format '%s' valgrind
==2084== Memcheck, a memory error detector
==2084== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2084== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==2084== Command: ./pacman -S --print --print-format %s valgrind
==2084==
==2084== Invalid read of size 1
==2084==    at 0x484D11D: strstr (vg_replace_strmem.c:1792)
==2084==    by 0x12620B: print_packages (util.c:1267)
==2084==    by 0x11F9DA: sync_prepare_execute (sync.c:817)
==2084==    by 0x11F550: sync_trans (sync.c:728)
==2084==    by 0x11FF72: pacman_sync (sync.c:965)
==2084==    by 0x11B5EB: main (pacman.c:1259)
==2084==  Address 0x65e61d0 is 0 bytes inside a block of size 3 free'd
==2084==    at 0x484412F: free (vg_replace_malloc.c:974)
==2084==    by 0x1261F2: print_packages (util.c:1264)
==2084==    by 0x11F9DA: sync_prepare_execute (sync.c:817)
==2084==    by 0x11F550: sync_trans (sync.c:728)
==2084==    by 0x11FF72: pacman_sync (sync.c:965)
==2084==    by 0x11B5EB: main (pacman.c:1259)
==2084==  Block was alloc'd at
==2084==    at 0x4841848: malloc (vg_replace_malloc.c:431)
==2084==    by 0x4A183DE: strdup (strdup.c:42)
==2084==    by 0x125ACB: print_packages (util.c:1198)
==2084==    by 0x11F9DA: sync_prepare_execute (sync.c:817)
==2084==    by 0x11F550: sync_trans (sync.c:728)
==2084==    by 0x11FF72: pacman_sync (sync.c:965)
==2084==    by 0x11B5EB: main (pacman.c:1259)
...
==2084== ERROR SUMMARY: 50 errors from 40 contexts (suppressed: 0 from 0)
```

Figure 1.3: Detecting the bug with Valgrind

Exploit Scenario

Pacman starts using multiple threads and uses the `print_packages` function in one thread and performs an allocation of a similar size to the freed `temp` variable in another thread with attacker-controlled content. The attacker leverages this fact to exploit the program by manipulating its heap memory through the vulnerable code path.

Recommendations

Short term, add an assignment of `temp = string;` after the temp variable is freed in the vulnerable code path in the `print_packages` function. This will prevent the use-after-free issue.

Long term, regularly scan the code with static analyzers like scan-build.

2. Null pointer dereferences	
Severity: **Informational**	Difficulty: Low
Type: Denial of Service	Finding ID: TOB-PACMAN-2
Target: ● `pacman/src/pacman/callback.c:656-660` ● `pacman/lib/libalpm/util.c:469-481`	

Description

The cb_progress function first checks if a `pkgname` is a null pointer in a ternary operator (1) and then may use that `pkgname` in order to format a string in (2) or (3) (figure 2.1). This leads to a crash if the `pkgname` is a null pointer.

The severity of this finding is informational since if the `cb_progress` function would be called with a null pointer, the program crash would be evident for the program users and developers.

```
void cb_progress(void *ctx, alpm_progress_t event, const char *pkgname,
                 int percent, size_t howmany, size_t current) {
...
len = strlen(opr) + ((pkgname) ? strlen(pkgname) : 0) + 2;      // <--- (1)
wcstr = calloc(len, sizeof(wchar_t));
/* print our strings to the alloc'ed memory */
#if defined(HAVE_SWPRINTF)
wclen = swprintf(wcstr, len, L"%s %s", opr, pkgname);           // <--- (2)
#else
/* because the format string was simple, we can easily do this without
 * using swprintf, although it is probably not as safe/fast. The max
 * chars we can copy is decremented each time by subtracting the length
 * of the already printed/copied wide char string. */
wclen = mbstowcs(wcstr, opr, len);
wclen += mbstowcs(wcstr + wclen, " ", len - wclen);
wclen += mbstowcs(wcstr + wclen, pkgname, len - wclen);         // <--- (3)
#endif
```

Figure 2.1: pacman/src/pacman/callback.c#L656-660

An additional case of null pointer dereference is present in the `_alpm_chroot_write_to_child()` function, if the `out_cb` argument is null.

```
typedef ssize_t (*_alpm_cb_io)(void *buf, ssize_t len, void *ctx);

// [...]

static int _alpm_chroot_write_to_child(alpm_handle_t *handle, int fd,
             char *buf, ssize_t *buf_size, ssize_t buf_limit,
             _alpm_cb_io out_cb, void *cb_ctx)
{
       ssize_t nwrite;

       if(*buf_size == 0) {
              /* empty buffer, ask the callback for more */
              if((*buf_size = out_cb(buf, buf_limit, cb_ctx)) == 0) {
                     /* no more to write, close the pipe */
                     return -1;
              }
       }
```
Figure 2.2: pacman/lib/libalpm/util.c#L469-481

Recommendations
Short term, fix the potential null pointer dereferences in the `cb_progress` and `_alpm_chroot_write_to_child` functions.

Long term, use static analysis tools to detect cases where pointers are dereferenced without a preceding null check.

3. Allocation failures can lead to memory leaks or null pointer dereferences

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PACMAN-3

Target:
- `src/pacman/conf.c`
- `PR 96: lib/libalpm/alpm_list.c`
- `lib/libalpm/be_sync.c`

Description
There are a few code paths where allocation failures can lead to further memory leaks or null pointer dereferences. Those are:

- If the `strdup(path)` allocation fails in the `setdefaults` function (1 and 2) (figure 3.1), then the memory pointed by `rootdir` (2) would be leaked. This is because the `SETDEFAULT` macro would enter its error path and return -1 (3), not freeing the previously allocated memory.
- The `alpm_list_equal_ignore_order` function added in PR 96 fails to check that the `calloc` function returns a non-null value (figure 3.2). If `calloc` were to return `NULL`, this would lead to a null pointer dereference later on in the function (line 534).
- In `_alpm_validate_filename`, the `strlen(filename)` can be called with a null pointer if the `READ_AND_STORE(pkg->filename)` execution fails to allocate memory through the `STRDUP` macro use (figure 3.3).

The severity of this finding is informational since if an allocation fails, the program would likely stop functioning properly as it would fail to allocate any more memory anyway.

The first part of this issue (pertaining to `conf.c`, rather than `alpm_list.c`) has been found with the scan-build static analyzer.

```
int setdefaults(config_t *c) {
       alpm_list_t *i;

#define SETDEFAULT(opt, val)          \
    if(!opt) {                        \
        opt = val;                    \
        if(!opt) { return -1; }       \                       // (3)
    }

       if(c->rootdir) {
            char* rootdir = strdup(c->rootdir);                 // (2)
            ...
            char path[PATH_MAX];
            if(!c->dbpath) {
                    snprintf(path, PATH_MAX, "%s/%s", rootdir, &DBPATH[1]);
                    SETDEFAULT(c->dbpath, strdup(path));     // (1)
            }
            if(!c->logfile) {
                    snprintf(path, PATH_MAX, "%s/%s", rootdir, &LOGFILE[1]);
                    SETDEFAULT(c->logfile, strdup(path));    // (1)
            }
```

Figure 3.1: pacman/src/pacman/conf.c#L1139-1153

```
int SYMEXPORT alpm_list_equal_ignore_order(const alpm_list_t *left,
                const alpm_list_t *right, alpm_list_fn_cmp fn)
{
        const alpm_list_t *l = left;
        const alpm_list_t *r = right;
        int *matched;

        if((l == NULL) != (r == NULL)) {
                return 0;
        }

        if(alpm_list_count(l) != alpm_list_count(r)) {
                return 0;
        }

        matched = calloc(alpm_list_count(right), sizeof(int));

        for(l = left; l; l = l->next) {
                int found = 0;
                int n = 0;

                for(r = right; r; r = r->next, n++) {
                        /* make sure we don't match the same value twice */
                        if(matched[n]) {
                                continue;
                        }
```

Figure 3.2: PR 96: lib/libalpm/alpm_list.c#L511-536

```
#define READ_AND_STORE(f) do { \
        READ_NEXT(); \
        STRDUP(f, line, goto error); \
} while(0)

#define STRDUP(r, s, action) do { \
        if(s != NULL) { \
                r = strdup(s); \
                if(r == NULL) { \
                  _alpm_alloc_fail(strlen(s)); \
                  action; \
       } } \
       else { r = NULL; } } \
while(0)

READ_AND_STORE(pkg->filename);
if(_alpm_validate_filename(db, pkg->name, pkg->filename) < 0) { ... }
```

Figure 3.3: pacman/lib/libalpm/be_sync.c#L591-595

Recommendations

Short term, fix the memory leaks or null pointer dereferences as detailed in this finding.

Long term, regularly scan the code with static analyzers like scan-build.

4. Buffer overflow read in string_length utility function

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PACMAN-4
Target: `src/pacman/util.c`	

Description

The `string_length` utility function (figure 4.1) skips ANSI color codes when computing the length. When a string includes the "`\033`" byte that starts the ANSI color code sequence but does not have the "`m`" character which ends it, the function will read memory past the end of the string, causing a buffer overflow read.

This can lead to a program crash or other issues, depending on how the function is used.

```
static size_t string_length(const char *s) {
	int len;
	wchar_t *wcstr;

	if(!s || s[0] == '\0') {
		return 0;
	}
	if(strstr(s, "\033")) {
		char* replaced = malloc(sizeof(char) * strlen(s));
		int iter = 0;
		for(; *s; s++) {
			if(*s == '\033') {
				while(*s != 'm') {
					s++;
				}
			} else {
				replaced[iter] = *s;
				iter++;
			}
		}
		replaced[iter] = '\0';
```

Figure 4.1: pacman/src/pacman/util.c#L452-473

Recommendations

Short term, fix the buffer overflow read issue in the `string_length` function.

Long term, implement a fuzzing harness for the `string_length` function to make sure it doesn't contain any bugs. An example harness code for it can be found in figure 4.2 and which can be compiled and run using the following commands:

```
clang -fsanitize=fuzzer,address main.c -ggdb -o fuzzer
./fuzzer
```

Figure 4.3 shows an example output of such a fuzzer. We also implemented this harness as part of the Pacman codebase as detailed in Appendix D.

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <wchar.h>

static size_t string_length(const char *s) { ... }

int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
       if (Size == 0) return 0;

       // Prepare a null terminated string
       char* x = malloc(Size+1);
       memcpy(x, Data, Size);
       x[Size] = 0;

       string_length(x);

       free(x);
       return 0;
}
```

Figure 4.2: Example fuzzing harness that uses libFuzzer to test the `string_length` function

```
$ clang -fsanitize=fuzzer,address main.c -ggdb -o fuzzer
$ ./fuzzer
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1790240281
INFO: Loaded 1 modules   (12 inline 8-bit counters): 12 [0x56046acc5fc0,
0x56046acc5fcc),
INFO: Loaded 1 PC tables (12 PCs): 12 [0x56046acc5fd0,0x56046acc6090),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 4 ft: 5 corp: 1/1b exec/s: 0 rss: 30Mb
...
#173    REDUCE cov: 5 ft: 6 corp: 2/2b lim: 4 exec/s: 0 rss: 31Mb L: 1/1 MS: 1
=================================================================
==2873139==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000006b53
at pc 0x56046ac84a76 bp 0x7ffd09e07ef0 sp 0x7ffd09e07ee8
READ of size 1 at 0x602000006b53 thread T0
    #0 0x56046ac84a75 in string_length /fuzz/main.c:21:11
    #1 0x56046ac8483d in LLVMFuzzerTestOneInput /fuzz/main.c:56:2
    #2 0x56046abad383 in fuzzer::Fuzzer::ExecuteCallback(unsigned char const*,
unsigned long) (/fuzz/fuzzer+0x3e383) (BuildId:
65f386451dc943b740358c52379831570eef52be)
…

0x602000006b53 is located 0 bytes to the right of 3-byte region
[0x602000006b50,0x602000006b53)
allocated by thread T0 here:
    #0 0x56046ac499fe in malloc (/fuzz/fuzzer+0xda9fe) (BuildId:
65f386451dc943b740358c52379831570eef52be)
    #1 0x56046ac847db in LLVMFuzzerTestOneInput /root/fuz/main.c:53:12
...

SUMMARY: AddressSanitizer: heap-buffer-overflow /root/fuz/main.c:21:11 in
string_length
...
```

Figure 4.3: Output from the fuzzer from figure 4.2

5. Undefined behavior or potential null pointer dereferences by passing null pointers to functions requiring non-null arguments

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PACMAN-5
Target: `multiple codepaths`	

Description

There are a few code paths where a null pointer dereference or undefined behavior may happen if certain conditions are met. Those issues can be detected with the scan-build static analyzer or by building and by running Pacman with the undefined behavior sanitizer. The scan-build results were shared along with this report.

One of the code paths found by scan-build is in the lib/libalpm/remove.c file. The `closedir(dir)` function may be called with a null pointer when the condition that calls `regcomp(...)` is true (figure 5.1). This is undefined behavior since the `closedir` function argument is marked as nonnull.

```
static void shift_pacsave(alpm_handle_t *handle, const char *file) {
        DIR *dir = NULL;
        ...
        if(regcomp(&reg, regstr, REG_EXTENDED | REG_NEWLINE) != 0) {
                goto cleanup;
        }

        dir = opendir(dirname); // <-- the dir was only modified here
        ...
cleanup:
        free(dirname);
        closedir(dir);
```

Figure 5.1: pacman/lib/libalpm/remove.c#L349-423

Another case is in the `mount_point_list` function (figure 5.2). If the STRDUP macro is executed with a null pointer `mnt->mnt_dir`, then the `strlen(mp->mount_dir)` call will take a null pointer.

```
static alpm_list_t *mount_point_list(alpm_handle_t *handle) {
        ...
#if defined(HAVE_GETMNTENT) && defined(HAVE_MNTENT_H)
        ...
        while((mnt = getmntent(fp))) {
              CALLOC(mp, 1, sizeof(alpm_mountpoint_t), RET_ERR(handle,
ALPM_ERR_MEMORY, NULL));
              STRDUP(mp->mount_dir, mnt->mnt_dir, free(mp); RET_ERR(handle,
ALPM_ERR_MEMORY, NULL));
              mp->mount_dir_len = strlen(mp->mount_dir);
```

Figure 5.2: pacman/lib/libalpm/diskspace.c#L95-116

In addition to that, figure 5.3 shows a run of pacman with undefined behavior sanitizer that detects other cases of this issue.

```
# CFLAGS=-fsanitize=address,undefined LDFLAGS=-fsanitize=address,undefined meson
setup sanitize
# cd sanitize
# CFLAGS=-fsanitize=address,undefined LDFLAGS=-fsanitize=address,undefined meson
compile
# ./pacman -Syuu
:: Synchronizing package databases...
 core downloading...
 extra downloading...
:: Starting full system upgrade...
../lib/libalpm/util.c:1149:9: runtime error: null pointer passed as argument 1,
which is declared to never be null
../lib/libalpm/util.c:1151:10: runtime error: null pointer passed as argument 1,
which is declared to never be null
../lib/libalpm/util.c:1192:4: runtime error: null pointer passed as argument 2,
which is declared to never be null
...
:: Proceed with installation? [Y/n] Y
...
```

Figure 5.3: Running Pacman with UndefinedBehavior sanitizer

Recommendation

Short term, fix the cases where functions marked with non-null arguments are called with null pointers.

Long term, regularly test pacman with undefined behavior sanitizer as well as scanning its codebase with static analyzers such as scan-build.

6. Undefined behavior from use of atoi

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PACMAN-6
Target: `lib/libalpm/be_local.c, src/pacman/pacman.c`	

Description

The `atoi` function is used to convert strings to integers, when parsing local database files and command line arguments (figure 6.1, 6.2). The behavior of `atoi` is undefined in the case that the inputted string is not a valid formatted number, or in the case of an overflow. The severity of this finding is informational since, in practice, `atoi` will typically return a dummy value, such as 0 or -1, in the case of an incorrect input or an overflow.

```
} else if(strcmp(line, "%REASON%") == 0) {
	READ_NEXT();
	info->reason = (alpm_pkgreason_t)atoi(line);
```

Figure 6.1: Use of `atoi` (lib/libalpm/be_local.c#L774-776)

```
case OP_ASK:
	config->noask = 1;
	config->ask = (unsigned int)atoi(optarg);
	break;
...
case OP_DEBUG:
	/* debug levels are made more 'human readable' than using a raw logmask
	 * here, error and warning are set in config_new, though perhaps a
	 * --quiet option will remove these later */
	if(optarg) {
		unsigned short debug = (unsigned short)atoi(optarg);
		switch(debug) {
			case 2:
				config->logmask |= ALPM_LOG_FUNCTION;
				__attribute__((fallthrough));
			case 1:
				config->logmask |= ALPM_LOG_DEBUG;
				break;
			default:
				pm_printf(ALPM_LOG_ERROR, _("'%s' is not a valid debug
level\n"),
						optarg);
				return 1;
		}
	} else {
```

```
                config->logmask |= ALPM_LOG_DEBUG;
        }
        /* progress bars get wonky with debug on, shut them off */
        config->noprogressbar = 1;
        break;
```

Figure 6.2: Uses of `atoi` (src/pacman/pacman.c#L382-430)

Recommendations

Short term, use the `strtol` function instead of `atoi`. Check the `errno` value after calling `strtol` to check for a failed conversion. Make sure to perform bounds checking when casting the `long` value returned by `strtol` down to an `int`.

7. Database parsers fail silently if an option is not recognized

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PACMAN-7
Target: `lib/libalpm/be_sync.c, lib/libalpm/be_local.c`	

Description

The `sync_db_read` and `local_db_read` functions, which are responsible for parsing sync database files and local database files respectively, fail silently if an option is not recognized. This can cause a configuration option to not be set which may cause issues if, for example, the local installation of Pacman is out of date and does not support newly-added configuration options.

Exploit Scenario

Support for SHA-3 hash verification is added, along with a corresponding configuration option `%SHA3SUM%`. Older installations of Pacman, which do not support this configuration option, will instead ignore it. This causes package hashes to not be verified.

Recommendations

Short term, add default behavior in the `sync_db_read` and `local_db_read` functions for when a configuration option is not recognized. Unrecognized options should cause a log message or an error.

8. Cache cleaning function may delete the wrong files

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PACMAN-8
Target: `src/pacman/sync.c`	

Description

In the `sync_cleancache` function, a path is constructed for deletion using the `snprintf` function. A maximum path length of `PATH_MAX` is given (on Linux, this value is 4096 characters). However, there is no check to ensure that the path created by `snprintf` was not cut short by the limit. This can lead to a different path than intended getting deleted.

The severity of this finding is informational since it is highly unlikely that Pacman would use a path this long in practice.

```
/* build the full filepath */
snprintf(path, PATH_MAX, "%s%s", cachedir, ent->d_name);

/* short circuit for removing all files from cache */
if(level > 1) {
	ret += unlink_verbose(path, 0);
	continue;
}
```

Figure 8.1: pacman/src/pacman/sync.c#L241-248

Recommendations

Short term, add a check which compares the value returned by `snprintf` and ensures that it is less than `PATH_MAX`.

9. Integer underflow in a length check leading to out-of-bounds read in alpm_extract_keyid

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PACMAN-9
Target: `lib/libalpm/signing.c`	

Description

The `alpm_extract_keyid` function (figure 9.1) contains an out-of-bounds read issue due to an integer underflow in `length_check` function when a specifically crafted input is provided (figure 9.2).

```
int SYMEXPORT alpm_extract_keyid(alpm_handle_t *handle, const char *identifier,
		const unsigned char *sig, const size_t len, alpm_list_t **keys) {
	size_t pos, blen, hlen, ulen;
	pos = 0;

	while(pos < len) {
		if(!(sig[pos] & 0x80)) { ... - return signature format error }

		if(sig[pos] & 0x40) {
			/* new packet format */
			if(length_check(len, pos, 1, handle, identifier) != 0) {
				return -1;
			}
			pos = pos + 1;
```

Figure 9.1: pacman/lib/libalpm/signing.c#L1101-1223

```
/* Check to avoid out of boundary reads */
static size_t length_check(size_t length, size_t position, size_t a,
		alpm_handle_t *handle, const char *identifier) {
	if( a == 0 || length - position <= a) {
		_alpm_log(handle, ALPM_LOG_ERROR,
				_("%s: signature format error\n"), identifier);
		return -1;
	} else {
		return 0;
	}
}
```

Figure 9.2: pacman/lib/libalpm/signing.c#L1043-1054

The `length_check` function is used to confirm if advancing a position (`pos`) index is safe. It is used by `alpm_extract_keyid` for example in the following way:

```
        length_check(len, pos, 2, handle, identifier)
```

The `len` is the length of the signature buffer (`sig`) and `pos` is an index in that buffer. However, the `pos` index can be bigger than the `len` variable and when that happens, then the `length-position` computation in the `length_check` function underflows and the function returns 0, leading to the out-of-bounds read.

We found this issue by fuzzing the `alpm_extract_keyid` function. The fuzzing harness code is included in Appendix D.

Recommendation
Short term, fix the integer underflow issue in the `length_check` function. This will prevent out-of-bound reads in the `alpm_extract_keyid` function.

Long term, fuzz the Pacman functions, for example as shown in Appendix D.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria

Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Remove the `if (fd >= 0)` condition in the `_alpm_pkg_load_internal` function since it is always true. This is because if the `fd` is less than 0 the function returns NULL in a previous condition.

```
alpm_pkg_t *_alpm_pkg_load_internal(alpm_handle_t *handle,
                const char *pkgfile, int full) {
        int ret, fd;
        ...
        fd = _alpm_open_archive(handle, pkgfile, &st, &archive, ALPM_ERR_PKG_OPEN);
        if(fd < 0)
                ...
                return NULL;
        }
        ...
error:
        _alpm_pkg_free(newpkg);
        _alpm_archive_read_free(archive);
        if(fd >= 0) {
                close(fd);
        }

        return NULL;
}
```

Figure C.1: pacman/lib/libalpm/be_package.c#L569-688

Use the `strdup` function to duplicate a string in the `clean_filename` function. This can be done instead of computing the string length, allocating memory and copying the filename with `memcpy`.

```
static char *clean_filename(const char *filename) {
        int len = strlen(filename);
        char *p;
        char *fname = malloc(len + 1);
        memcpy(fname, filename, len + 1);
```

Figure C.2: pacman/src/pacman/callback.c#L755-760

Refactor the dead assignment to the `curlerr` variable in the curl_check_finished_download function. The assignment should be either removed or there should be code that would act upon its value.

```
static int curl_check_finished_download(alpm_handle_t *handle, CURLM *curlm, CURLMsg
*msg, const char *localpath, int *active_downloads_num) {
	...
	CURLcode curlerr;
	...
		case CURLE_ABORTED_BY_CALLBACK:
			/* handle the interrupt accordingly */
			if(dload_interrupted == ABORT_OVER_MAXFILESIZE) {
				curlerr = CURLE_FILESIZE_EXCEEDED;
				payload->unlink_on_fail = 1;
				handle->pm_errno = ALPM_ERR_LIBCURL;
				_alpm_log(handle, ALPM_LOG_ERROR,
						_("failed retrieving file '%s' from %s :
expected download size exceeded\n"),
						payload->remote_name, hostname);
				server_soft_error(handle, payload->fileurl);
			}
			goto cleanup;
...
cleanup:
	... // <-- code that does not use the curlerr variable
	return ret;
}
```

Figure C.3: pacman/lib/libalpm/dload.c#L535-546

Remove the r variable from the `_cache_mtree_open` function and an assignment to it since it is unused. Alternatively, if it is intended, use the value of r within the if condition.

```
static struct archive *_cache_mtree_open(alpm_pkg_t *pkg) {
	int r;
	...
	if((r = _alpm_archive_read_open_file(mtree, mtfile, ALPM_BUFFER_SIZE))) {
		_alpm_log(pkg->handle, ALPM_LOG_ERROR, _("error while reading file %s:
%s\n"),
					mtfile, archive_error_string(mtree));
		_alpm_archive_read_free(mtree);
		GOTO_ERR(pkg->handle, ALPM_ERR_LIBARCHIVE, error);
	}

	free(mtfile);
	return mtree;

error:
	free(mtfile);
	return NULL;
}
```

Figure C.4: pacman/lib/libalpm/be_local.c#L251-284

Return an error if the call to `malloc` fails in `alpm_list_add_sorted` function. Currently, the function returns the existing list, even though it failed to insert the element as expected. This function is currently unused, so this does not yet pose a security concern.

```
add = malloc(sizeof(alpm_list_t));
if(add == NULL) {
	return list;
}
```

Figure C.5: pacman/lib/libalpm/alpm_list.c#L115-118

Restore the `list` variable to its original state before returning in the `alpm_list_reverse` function. In the beginning of the function, the `list->prev` member is backed up and then modified. However, in the case of an error, this backup is not restored, leaving the list in an invalid state.

```
alpm_list_t SYMEXPORT *alpm_list_reverse(alpm_list_t *list) {
	const alpm_list_t *lp;
	alpm_list_t *newlist = NULL, *backup;

	if(list == NULL) {
		return NULL;
	}

	lp = alpm_list_last(list);
	/* break our reverse circular list */
	backup = list->prev;
	list->prev = NULL;

	while(lp) {
		if(alpm_list_append(&newlist, lp->data) == NULL) {
			alpm_list_free(newlist);
			return NULL;
		}
		lp = lp->prev;
	}
	list->prev = backup; /* restore tail pointer */
	return newlist;
}
```

Figure C.6: pacman/lib/libalpm/alpm_list.c#L403-426

Rename the `type` variable to `event` in the `alpm_list_reverse` function. When a download payload is sent over a pipe in PR 23 (from the `_alpm_sandbox_cb_dl` function to the `_alpm_sandbox_process_cb_download` function), a variable called `event` is sent through the pipe and received into a variable called `type`. This can cause confusion when reading the sending and receiving code.

Rework the `had_error` variable in the `curl_download_internal_sandboxed` function. The variable will always be set to `true` by the time the loop shown in figure C.7 exits. This is because every `break` statement is accompanied with a statement setting `had_error` to `true`. This means that the variable does not track any useful information.

```
bool had_error = false;
while(true) {
	_alpm_sandbox_callback_t callback_type;
	ssize_t got = read(callbacks_fd[0], &callback_type, sizeof(callback_type));
	if(got < 0 || (size_t)got != sizeof(callback_type)) {
		had_error = true;
		break;
	}

	if(callback_type == ALPM_SANDBOX_CB_DOWNLOAD) {
		if(!_alpm_sandbox_process_cb_download(handle, callbacks_fd[0])) {
			had_error = true;
			break;
		}
	}
	else if(callback_type == ALPM_SANDBOX_CB_LOG) {
		if(!_alpm_sandbox_process_cb_log(handle, callbacks_fd[0])) {
			had_error = true;
			break;
		}
	}
}


if(had_error) {
	kill(pid, SIGTERM);
}
```

Figure C.7: PR 23: pacman/lib/libalpm/dload.c#L974-1000

Verify the `%REASON%` field before casting it to an `alpm_pkgreason_t` enum value in the `local_db_read` function. Otherwise, the field may contain a value which is a valid integer but not a valid `alpm_pkgreason_t` value.

```
} else if(strcmp(line, "%REASON%") == 0) {
	READ_NEXT();
	info->reason = (alpm_pkgreason_t)atoi(line);
```

Figure C.8: pacman/lib/libalpm/be_local.c#L774-776

Correct the log message at the end of the `curl_download_internal` function. The message incorrectly states the value returned by the function.

```
_alpm_log(handle, ALPM_LOG_DEBUG, "curl_download_internal return code is %d\n",
err);
return err ? -1 : updated ? 0 : 1;
```

Figure C.9: pacman/lib/libalpm/dload.c#L937-938

Refactor the ALPM public functions from returning an `int` to return a status type. This new type could be a typedef for an `int`. Such a change would make it easier to perform static analysis to find all functions that return the typedef and ensure that the callers check for errors.

```
[Errors]

The library provides a global variable pm_errno.
It aims at being to the library what errno is for C system calls.

Almost all public library functions are returning an integer value: 0
indicating success, -1 indicating a failure.
If -1 is returned, the variable pm_errno is set to a meaningful value
Wise frontends should always care for these returned values.

Note: the helper function alpm_strerror() can also be used to translate one
specified error code into a more friendly sentence, and alpm_strerrorlast()
does the same for the last error encountered (represented by pm_errno).
```

Figure C.10: pacman/README?plain=1#L144-156

D. Fuzzing Pacman code

During the audit, Trail of Bits used fuzzing, an automated testing technique in which code paths are executed with random data to find bugs resulting from the incorrect handling of unexpected data. For this, we used libFuzzer, an in-process coverage-guided fuzzer, and we extended the Pacman build system with new executables to fuzz certain code paths. This helped us to find issues detailed in findings TOB-PACMAN-4 and TOB-PACMAN-9.

We implemented fuzzing harnesses for:

- The `string_length` function
- The `wordsplit` function
- Parsing of config files through the `parseconfigfile` function
- The extraction of keys from signature data through the `alpm_extract_keyid` function

For this, we also modified the `meson.build` file so that all the files are built with AddressSanitizer (`-fsanitize=address` compiler and linker flag) that helps detect more bugs. In order to build the harnesses and run them, we leveraged the following commands:

```
CC=clang meson setup fuzz
cd fuzz
CC=clang meson compile <harness, e.g., fuzz_alpm_extract_keyid>
./<harness binary>
```

We used the clang compiler because in our case, where we performed fuzzing in an Arch Linux docker container, the GCC compiler did not support the `-fsanitize=fuzzer` flag that enables the libFuzzer fuzzing framework.

The implemented code can be seen in figure D.1 and will also be sent as a merge request against the Pacman repository after the final readout of this report.

Fuzzing harness notes

Below we present some notes about the changes and harnesses we developed.

- The `add_project_arguments` added to the `meson.build` is suboptimal and has to be refactored, so it is enabled only when fuzzing harnesses are built, or the specific dependencies/libraries need to have separate fuzzing targets so they are built with AddressSanitizer enabled.
- None of the external dependencies are built with AddressSanitizer or UndefinedBehavior sanitizer. This may cause false positive crashes when new harnesses are developed that leverage the code paths of those dependencies, or it may lead to not detecting valid bugs.

- We encountered some issues with including headers from `src/pacman` in `fuzz_parseconfigfile` and `fuzz_string_length` harnesses, which we worked around by providing the fuzzed function declarations in the harnesses itself. This should be fixed so that the headers are included properly. The same goes for, e.g., the `extern void *config;` global variable.
- The `fuzz_wordsplit` harness can be refactored to free its resources via the `wordsplit_free` function.
- The `fuzz_alpm_extract_keyid` does not set proper handle or filename arguments. Setting these arguments may leverage more code paths in the harness.
- The `fuzz_parseconfigfile` is far from ideal: the generated input may include other files from the filesystem to be parsed by the code, which is nondeterministic. The solution to that could be:
    - Either use `chroot` or mount namespaces so that the fuzzer works in an isolated filesystem with no other files included,
    - Or changing the harness so it only generates semi-valid config files.
- We added an `#ifndef FUZZING_PACMAN` to remove the `main` function of Pacman for the fuzzing harnesses which need the src/pacman code. Otherwise, the linking of the harness would fail due to multiple definitions of the `main` symbol.

Recommendations and further work

Going further, we recommend the Pacman team to:

- Refactor the build system to better support the building of fuzzing harnesses (instead of setting global arguments as we did).
- Extend the build system so it also builds all of the dependencies' code with sanitizers enabled.
- Test and fuzz the code with other sanitizers enabled that we haven't tried here (e.g., MemorySanitizer or ThreadSanitizer in case threads would ever be used in Pacman).
- Implementing more fuzzing harnesses, for example for the `dload_parseheader_cb` function and other functionalities that parse untrusted data.
- Fuzzing Pacman continuously with each release. This can be done by integrating it into the oss-fuzz project, which allows for free fuzzing of open source projects. However, please note that the company beyond the oss-fuzz project, Google, will know about the found vulnerabilities first.

```
diff --git a/meson.build b/meson.build
index 43705338..bfeca3af 100644
--- a/meson.build
+++ b/meson.build
@@ -14,6 +14,8 @@ libalpm_version = '13.0.1'

 cc = meson.get_compiler('c')

+add_project_arguments(['-fsanitize=address', '-fno-omit-frame-pointer', '-ggdb', '-O0'], language : 'c')
```

```
+
 # commandline options
 PREFIX = get_option('prefix')
 DATAROOTDIR = join_paths(PREFIX, get_option('datarootdir'))
@@ -305,6 +307,8 @@ subdir('src/pacman')
 subdir('src/util')
 subdir('scripts')

+subdir('src/fuzzing')
+
 # Internationalization
 if get_option('i18n')
   i18n = import('i18n')
@@ -396,6 +400,45 @@ executable(
   install : true,
 )

+# Note: fuzz targets below must be built with Clang compiler for the -fsanitize=fuzzer flag
+executable(
+    'fuzz_wordsplit',
+    fuzz_wordsplit_sources,
+    include_directories : includes,
+    link_with : [libcommon],
+    dependencies : [],
+    c_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer'],
+    link_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer'],
+)
+
+executable(
+    'fuzz_string_length',
+    [fuzz_string_length_sources, pacman_sources],
+    include_directories : includes,
+    link_with : [libalpm_a, libcommon],
+    dependencies : [],
+    c_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer', '-DFUZZING_PACMAN'],
+    link_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer'],
+)
+executable(
+    'fuzz_alpm_extract_keyid',
+    [fuzz_alpm_extract_keyid_sources, pacman_sources],
+    include_directories : includes,
+    link_with : [libalpm_a, libcommon],
+    dependencies : [],
+    c_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer', '-DFUZZING_PACMAN'],
+    link_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer'],
+)
+executable(
+    'fuzz_parseconfigfile',
+    [fuzz_parseconfigfile_sources, pacman_sources],
+    include_directories : includes,
+    link_with : [libalpm_a],
+    dependencies : [],
+    c_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer', '-DFUZZING_PACMAN'],
+    link_args : ['-fsanitize=fuzzer,address', '-ggdb', '-O0', '-fno-omit-frame-pointer'],
+)
+
 foreach wrapper : script_wrappers
   cdata = configuration_data()
   cdata.set_quoted('BASH', BASH.full_path())
diff --git a/src/fuzzing/fuzz_alpm_extract_keyid.c b/src/fuzzing/fuzz_alpm_extract_keyid.c
new file mode 100644
index 00000000..febbd57a
--- /dev/null
+++ b/src/fuzzing/fuzz_alpm_extract_keyid.c
@@ -0,0 +1,26 @@
+#define _XOPEN_SOURCE
+#include <stdio.h>
+#include <stdlib.h>
```

```
+#include <stdint.h>
+#include <string.h>
+#include <wchar.h>
+
+/* libalpm */
+#include "alpm.h"
+#include "alpm_list.h"
+#include "handle.h"
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size);
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
+    if (Size == 0)
+        return 0;
+
+    alpm_handle_t handle;                // TODO/FIXME?
+    const char* filename = "/dev/null"; // TODO/FIXME?
+
+    alpm_list_t *keys = NULL;
+    alpm_extract_keyid(&handle, filename, /* sig */ Data, /* len */ Size, &keys);
+
+    return 0;
+}
diff --git a/src/fuzzing/fuzz_parseconfigfile.c b/src/fuzzing/fuzz_parseconfigfile.c
new file mode 100644
index 00000000..4746141d
--- /dev/null
+++ b/src/fuzzing/fuzz_parseconfigfile.c
@@ -0,0 +1,43 @@
+#include <stdio.h>
+#include <stdlib.h>
+#include <stdint.h>
+#define _GNU_SOURCE
+#include <sys/mman.h>
+#include <unistd.h>
+
+// TODO/FIXME: Fix the util.h include
+//#include "conf.h"
+// And remove that function header from here
+int parseconfigfile(const char *s);
+extern void *config;
+void *config_new(void);
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size);
+
+// TODO/FIXME: This fuzzer should always be run from a chroot
+// without any other files in it; otherwise the configfile may refer
+// to other files
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
+    static void* config_object = 0;
+
+    // TODO/FIXME: The harness needs to be run with -detect_leaks=0
+    // because the config object here is detected as a leak
+    if (!config_object) {
+        config = config_object = config_new();
+    }
+
+    if (Size == 0)
+        return 0;
+
+    int fd = memfd_create("input", 0); // create an in-memory file we can have path to
+    write(fd, Data, Size);
+
+    char path[64] = {0};
+    sprintf(path, "/proc/self/fd/%d", fd);
+
+    parseconfigfile(path);
+
+    close(fd);
+
+    return 0;
+}
diff --git a/src/fuzzing/fuzz_string_length.c b/src/fuzzing/fuzz_string_length.c
new file mode 100644
index 00000000..8991b476
--- /dev/null
+++ b/src/fuzzing/fuzz_string_length.c
@@ -0,0 +1,26 @@
+#include <stdio.h>
+#include <stdlib.h>
+#include <string.h>
+
+// TODO/FIXME: Fix the util.h include
+//#include "util.h"
+// And remove that function header from here
+size_t string_length(const char *s);
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size);
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
+    if (Size == 0)
+        return 0;
+
+    // Prepare a null terminated string
+    char* cstring = malloc(Size+1);
+    memcpy(cstring, Data, Size);
+    cstring[Size] = 0;
+
+    string_length(cstring);
+
+    free(cstring);
+
+    return 0;
+}
diff --git a/src/fuzzing/fuzz_wordsplit.c b/src/fuzzing/fuzz_wordsplit.c
new file mode 100644
index 00000000..e2e10210
--- /dev/null
+++ b/src/fuzzing/fuzz_wordsplit.c
@@ -0,0 +1,36 @@
+#define _XOPEN_SOURCE
+#include <stdio.h>
+#include <stdlib.h>
+#include <stdint.h>
+
+#include "util-common.h"
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size);
+
+int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
+    if (Size == 0)
+        return 0;
+
+    // Prepare a null terminated string
+    char* cstring = malloc(Size+1);
+    memcpy(cstring, Data, Size);
+    cstring[Size] = 0;
+
+    char** ptr = wordsplit(cstring);
+
+    // Free the memory allocated by wordsplit
+    if (ptr) {
+        int i = 0;
+        char* p = ptr[i++];
+        while (p) {
+            free(p);
+               p = ptr[i++];
+           }
+           free(ptr);
+       }
+
+       // Free the allocated cstring
+       free(cstring);
+
+       return 0;
+}
diff --git a/src/fuzzing/meson.build b/src/fuzzing/meson.build
new file mode 100644
index 00000000..9a8555c2
--- /dev/null
+++ b/src/fuzzing/meson.build
@@ -0,0 +1,15 @@
+fuzz_wordsplit_sources = files('''
+  fuzz_wordsplit.c
+'''.split())
+
+fuzz_string_length_sources = files('''
+  fuzz_string_length.c
+'''.split())
+
+fuzz_alpm_extract_keyid_sources = files('''
+  fuzz_alpm_extract_keyid.c
+'''.split())
+
+fuzz_parseconfigfile_sources = files('''
+  fuzz_parseconfigfile.c
+'''.split())
\ No newline at end of file
diff --git a/src/pacman/pacman.c b/src/pacman/pacman.c
index e5c6e420..77c88392 100644
--- a/src/pacman/pacman.c
+++ b/src/pacman/pacman.c
@@ -1079,6 +1079,7 @@ static void cl_to_log(int argc, char *argv[])
         }
 }

+#ifndef FUZZING_PACMAN
 /** Main function.
  * @param argc
  * @param argv
@@ -1273,3 +1274,4 @@ int main(int argc, char *argv[])
         /* not reached */
         return EXIT_SUCCESS;
 }
+#endif //FUZZING_PACMAN
diff --git a/src/pacman/util.c b/src/pacman/util.c
index 5d42a6e9..a41c9e5e 100644
--- a/src/pacman/util.c
+++ b/src/pacman/util.c
@@ -449,7 +449,7 @@ static char *concat_list(alpm_list_t *lst, formatfn fn)
         return output;
 }

-static size_t string_length(const char *s)
+size_t string_length(const char *s)
 {
         int len;
         wchar_t *wcstr;
diff --git a/src/pacman/util.h b/src/pacman/util.h
index 52e79915..d8f7f5f2 100644
--- a/src/pacman/util.h
+++ b/src/pacman/util.h
@@ -47,6 +47,7 @@ typedef struct _pm_target_t {
         int is_explicit;
 } pm_target_t;

+size_t string_length(const char *s);
 void trans_init_error(void);
 /* flags is a bitfield of alpm_transflag_t flags */
 int trans_init(int flags, int check_valid);
```

Figure D.1: The diff for the fuzzing harness code

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From March 4 to Month 6, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Arch Linux team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 9 issues described in this report, Arch Linux has resolved 7 issues, and has partially resolved 2 issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Use-after-free vulnerability in the print_packages function	**Resolved**
2	Null pointer dereferences	**Resolved**
3	Allocation failures can lead to memory leaks or null pointer dereferences	**Resolved**
4	Buffer overflow read in string_length utility function	**Resolved**
5	Undefined behavior or potential null pointer dereferences by passing null pointers to functions requiring non-null arguments	**Partially Resolved**
6	Undefined behavior from use of atoi	**Resolved**
7	Database parsers fail silently if an option is not recognized	**Resolved**
8	Cache cleaning function may delete the wrong files	**Partially Resolved**
9	Integer underflow in a length check leading to out-of-bounds read in alpm_extract_keyid	**Resolved**

Detailed Fix Review Results

TOB-PACMAN-1: Use-after-free vulnerability in the print_packages function
Resolved in commit 36fcff6e. This commit adds an assignment which overwrites the freed temp variable with the newly allocated string variable.

TOB-PACMAN-2: Null pointer dereferences
Resolved in commit 74deada5. This commit adds the necessary checks to determine whether or not the pkgname variable is null before using it.

The Pacman developers correctly identified that the write_to_child function can only ever be called with a non-null callback, so a fix for that portion of the issue was not necessary.

TOB-PACMAN-3: Allocation failures can lead to memory leaks or null pointer dereferences
Resolved in commits 6711d10f and abc6dd74. Commit 6711d10f adds a check to the setdefaults function which ensures that the pointer returned strdup is non-null before using it. Commit abc6dd74 adds a check to the alpm_list_cmp_unsorted function which ensures that the pointer returned by calloc is non-null before using it.

The Pacman developers identified the code in figure 3.3 as not being an issue. We have confirmed that this is the case: it should not be possible for the line variable to be null without the goto error statement being executed; this prevents pkg->filename from being null in the call to the _alpm_validate_filename function.

TOB-PACMAN-4: Buffer overflow read in string_length utility function
Resolved in commit c9c56be3. This commit changes the string_length function so that it loops under more strict conditions: it stops once it reaches a character that isn't a digit or a semicolon, rather reading until an 'm' is found.

TOB-PACMAN-5: Undefined behavior or potential null pointer dereferences by passing null pointers to functions requiring non-null arguments
Partially resolved in commits f996f301 and ce528a26. Commit f996f301 adds a check to the shift_pacsave function which ensures that the dir pointer is non-null before using it in a closedir(dir) call. Commit ce528a26 adds a check to the mount_point_list function which ensures that the mnt->mnt_dir value is non-null before attempting to duplicate it into mp->mount_dir using the STRDUP macro. This ensures that mp->mount_dir will be non-null as well, which prevents undefined behavior during the call to strlen(mp->mount_dir).

The instances of undefined behavior shown in figure 5.3 have not been resolved.

TOB-PACMAN-6: Undefined behavior from use of atoi
Resolved in commit 6e6d3f18 and PR 136. Commit 6e6d3f18 replaces the use of `atoi` in the `_alpm_local_db_pkgpath` function with a set of `strcmp` comparisons. PR 136 replaces the uses of `atoi` in the `parsearg_global` function with calls to `strtol`, performing all the necessary error checks.

TOB-PACMAN-7: Database parsers fail silently if an option is not recognized
Resolved in commit e1dc6099. This commit adds a warning message which is logged in the case of an unknown option.

TOB-PACMAN-8: Cache cleaning function may delete the wrong files
Partially resolved in commit a6b25247. This commit adds a check determining whether `len > PATH_MAX`, and skipping the current file if this is the case. However, the check should instead determine whether `len >= PATH_MAX`, since the value returned by the `snprintf` function does not count the trailing null character.

In addition, the commit also fixes a very similar issue in the `sync_cleandb` function which was not found during the audit. However, the fix for the `sync_cleandb` function only prints an error message in the case of a problem, but does not skip the current file. In addition, the fix has the same issue mentioned above of using the > operator rather than the >= operator.

TOB-PACMAN-9: Integer underflow in a length check leading to out-of-bounds read in alpm_extract_keyid
Resolved in commit 16a2a797. This commit adds an additional check for `position > length` before computing `length - position`. If `position` is greater than `length`, an error is returned.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.