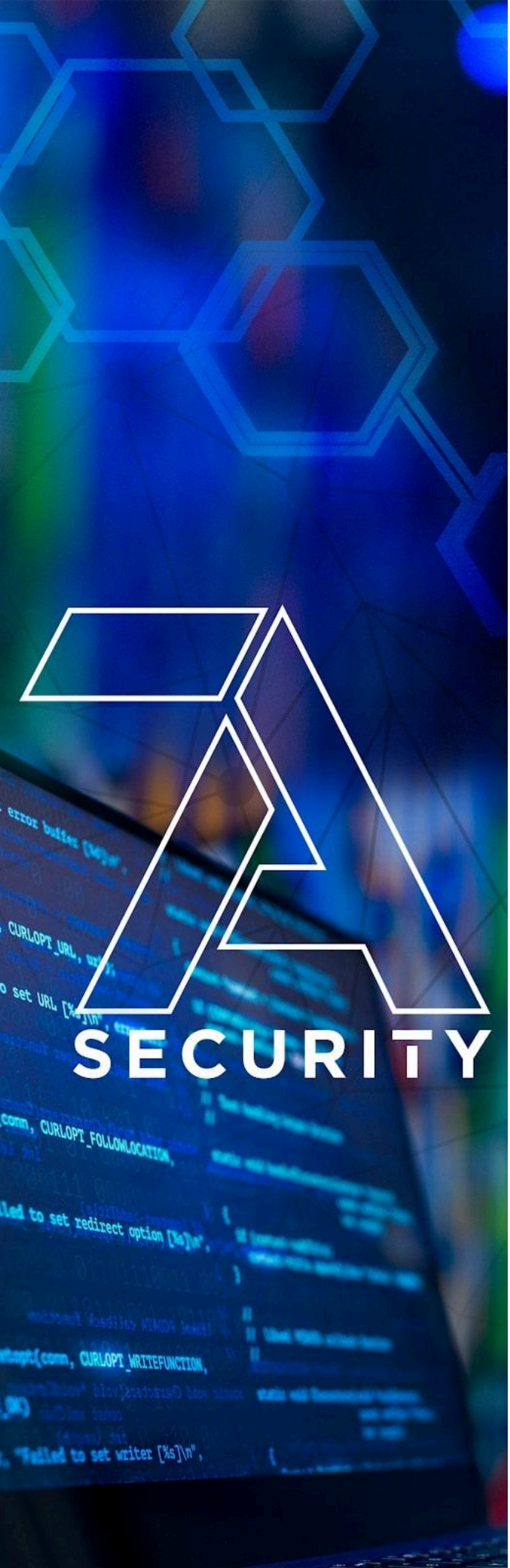

V2Ray Test Targets:

V2Ray Go Clients
V2Ray Go Servers
V2Ray Runtime
V2Ray Supply Chain

Pentest Report

Client:
V2Ray Team

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Miroslav Štampar, PhD.
- Stefan Nicula, PhD.

7ASecurity
Protect Your Site & Apps
From Attackers
sales@7asecurity.com
7asecurity.com

INDEX

Introduction

"A platform for building proxies to bypass network restrictions."
From https://github.com/v2fly/v2ray-core

This document outlines the results of a penetration test and *whitebox* security review conducted against V2Ray. The project was funded by the Open Technology Fund (OTF), and executed by 7ASecurity in March 2024. The audit team dedicated 32 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was initially expected to be easier during this assignment, as more vulnerabilities are typically identified and resolved after each testing cycle.

Please note that given the large size of the codebase, this audit focused on the most important components and most commonly used features of the V2Ray community.

During this iteration the goal was to review the V2Ray project as thoroughly as possible, to ensure users can be provided with the best possible security and privacy.

The methodology implemented was *whitebox*: 7ASecurity was provided with access to a reference server, documentation and source code. A team of 4 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by February 2024, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Telegram channel. The V2Ray team was helpful and responsive at all times, which facilitated the test for 7ASecurity, without introducing any unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items in the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: Whitebox Tests against V2Ray, Servers and Clients, written in Go
- WP2: Whitebox Tests against V2Ray Server Runtime Analysis via SSH
- WP3: Whitebox Tests against V2Ray Supply Chain Implementation

The findings of the security audit (WP1-2) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
3	*7*	*10*

Please note that the analysis of the remaining work package (WP3) is provided separately, in the following section of this report:

- WP3: V2Ray Supply Chain Implementation

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required. Additionally, it provides mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance. This includes insights related to the context, preparation, and general impressions gained throughout this test. Additionally, it offers a summary of the perceived security posture of the V2Ray project.

Scope

The following list outlines the items in scope for this project:

- **WP1: Whitebox Tests against V2Ray, Servers and Clients, written in Go**
  - Audited Source Code: https://github.com/v2fly/v2ray-core
- **WP2: Whitebox Tests against V2Ray Server Runtime Analysis via SSH**
  - As above
- **WP3: Whitebox Tests against V2Ray Supply Chain Implementation**
  - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *V2R-01-001*) for ease of reference and offers an estimated severity in brackets alongside the title.

V2R-01-006 WP1: Possible Fingerprint via Insecure Defaults *(Medium)*

Retest Notes: This issue is non-trivial to fix as it depends on the golang implementation. The V2Ray team will mitigate this weakness by educating users via documentation.

The *handlePlainHTTP* V2Ray method sets the *User-Agent* header to an empty string if the client does not provide one. While possibly intentional to prevent setting default *User-Agent* headers, this increases the odds for fingerprinting attacks by making it distinct from other HTTP clients. Confirmation was as follows:

PoC Code (file poc-request.py):

```
#!/usr/bin/python3
import requests

# Define the proxy address
proxy_address = 'http://192.168.0.25:1080'  # Change this to your proxy address

# Define the target URL
url = 'http://7as.es/s/v2ray-dani'

# Set up the proxy
proxies = {
   'http': proxy_address,
   'https': proxy_address
}

headers = {"User-Agent": "" }

# Send HTTP GET request through proxy
try:
   response = requests.get(url, proxies=proxies, headers=headers)
   print("Response from server:")
   print(response.text)
except requests.exceptions.RequestException as e:
   print("Error:", e)
```

Command:

```
python poc-request.py
```

Please note that the request received does not contain the *User-Agent* header.

Output (Headers without User-Agent):

```
Headers:

_____________________________________________
[Host] => 7as.es
[Accept] => */*
[Accept-Encoding] => gzip, deflate
[Connection] => close
```

The root cause of this issue appears to be in the following code path:

Affected File:

https://github.com/v2fly/v2ray-core/blob/.../proxy/http/server.go#L216

Affected Code:

```
func (s *Server) handlePlainHTTP(ctx context.Context, request *http.Request, writer
io.Writer, dest net.Destination, dispatcher routing.Dispatcher) error {
   [...]

   // Prevent UA from being set to golang's default ones
   if request.Header.Get("User-Agent") == "" {
      request.Header.Set("User-Agent", "")
   }
```

It is recommended to implement a default *User-Agent* behavior that is more difficult to fingerprint. For example:

- Set a default *User-Agent* header: Instead of setting the *User-Agent* header to an empty string, a more generic but non-identifiable *User-Agent* header could be provided. This ensures that the server is not easily distinguishable from other HTTP clients, reducing the risk of fingerprinting attacks.
- Randomize the *User-Agent* header: The *User-Agent* header may be randomized for each request. This increases the difficulty to profile or identify the server, based on the *User-Agent* header alone.
- Allow configuration options: Users could have options to configure custom *User-Agent* headers as needed. For example, users could set a specific or a random-per-request *User-Agent* parameter. Other possible enhancements in this area could be user-defined lists of user agents to rotate.

V2R-01-007 WP1/2: Possible V2Ray DoS via JA3 Fingerprints *(Medium)*

JA3 is a fingerprinting technique to identify and classify SSL/TLS clients based on the peculiarities of their TLS handshake messages[1]. This process generates a distinctive fingerprint for each software or device based on its handshake parameters, cryptographic algorithms, and supported extensions, differentiating TLS clients like web browsers and VPNs. Censorship authorities utilize JA3 signatures[2] to detect and block internet users, particularly targeting VPN traffic, which is used to bypass censorship via encrypted internet traffic. By monitoring TLS handshakes for known VPN JA3 fingerprints and maintaining a database of these signatures, authorities may selectively block VPN connections (i.e. V2Ray), effectively thwarting attempts to evade censorship. This strategy allows for targeted censorship of VPNs without impacting all TLS traffic.

OSINT research showed that the *3fe[...]c24* JA3 hash, linked to older V2Ray versions, matches previously documented software[3], indicating that this hash alone cannot definitively identify V2Ray due to possible matches with other network applications compiled with a similar Go compiler version. However, when combined with specific characteristics (V2R-01-006, V2R-01-008), it allows for the identification of a networked V2Ray client. Additionally, using known V2Ray JA3 fingerprints can block V2Ray usage effectively with minimal impact on legitimate traffic.

PoC:
During the proof of concept creation, various Linux 64-bit versions from v5.8.0[4] to v5.15.1[5] were tested against public V2Ray servers[6], all generating the identical JA3 fingerprint *7a15285d4efc355608b304698cd7f9ab*. Earlier versions from v5.0.7 to v5.2.1 produced a different fingerprint, *3fed133de60c35724739b913924b6c24*. The tests utilized the Python tool *pyja3*[7] and *Wireshark*[8], both confirming these fingerprints across all tests, including attempts with a single proxy request over V2Ray sessions. Despite testing different versions and platforms, JA3 fingerprints remained consistent across connection attempts and minor releases, indicating that V2Ray TLS communication can be easily identified using JA3 fingerprinting. This facilitates the straightforward creation of a censor database using the fingerprints of public releases.

Command:
```
ja3 capture.pcap
```

[1] https://github.com/salesforce/ja3
[2] https://github.com/net4people/bbs/issues/153
[3] https://ghost.security/resources/blog/attackers-guide-to-evading-honeypots-part1
[4] https://github.com/v2fly/v2ray-core/releases/tag/v5.8.0
[5] https://github.com/v2fly/v2ray-core/releases/tag/v5.15.1
[6] https://sshocean.com/v2ray/vmess
[7] https://pypi.org/project/pyja3/
[8] https://www.wireshark.org/

Output:


```
[
    {
        "destination_ip": "5.181.21.202",
        "destination_port": 443,
        "ja3":
"771,49195-49199-49196-49200-52393-52392-49161-49171-49162-49172-156-157-47-53-49170-10
-4865-4866-4867,0-5-10-11-13-65281-23-16-18-43-51,29-23-24-25,0",
        "ja3_digest": "7a15285d4efc355608b304698cd7f9ab",
        "source_ip": "192.168.0.107",
        "source_port": 57696,
        "timestamp": 1711013679.505045
    }
]
```


The root cause for this issue is that V2Ray uses the Go *crypto/tls* package for TLS without modifications[9], suggesting its TLS communications resemble typical Go clients based on the Go compiler version.

It is recommended to implement JA3 fingerprint randomization, to evade censorship targeting specific TLS fingerprints. This involves changing the JA3 signature of VPN clients or TLS applications periodically. Such strategy, including TLS ClientHello Extension Permutation -where extension order varies except for *pre_shared_key*- increases VPN resilience against censorship by masking VPN traffic as various TLS clients, complicating censor detection and database creation. Implemented in Google Chrome since early 2023[10], this aids in evading censorship and accessing the unrestricted internet, challenging authorities to maintain an up-to-date fingerprint database.

The proposed solution, which is already implemented[11], employs the uTLS package[12], derived from *crypto/tls*, which resists *ClientHello* fingerprinting, allows low-level access to the handshake process, enables the generation of fake session tickets, and provides additional functionalities. This approach can produce a unique JA3 fingerprint for each execution[13].

[9] https://github.com/v2fly/v2ray-core/blob/…/transport/internet/tls/tls.go#L1-L78
[10] https://www.fastly.com/blog/a-first-look-at-chromes-tls-clienthello-permutation-in-the-wild
[11] https://www.v2fly.org/en_US/v5/config/stream.html#utls
[12] https://github.com/refraction-networking/utls
[13] https://segmentfault.com/a/1190000041699815/en

V2R-01-008 WP1: Identification of V2Ray Servers via Keep-Alive *(Medium)*

It was found that V2Ray servers violate RFC2616[14], failing to remove *Keep-Alive* headers. Due to this behavior, headers like *Hop-By-Hop* and *Keep-Alive*, may make the proxy server leak information, facilitating fingerprinting. This issue may be confirmed as follows:

PoC Code (poc.go):

```
package main

import (
   "fmt"
   "net/http"
   "net/url"
   "os"
)

func main() {

   proxyAddress := "http://192.168.0.25:1080" # Change to your v2ray proxy

   target := "http://7as.es/s/v2ray-dani-keep-alive"

   proxyURL, err := url.Parse(proxyAddress)
   if err != nil {
       fmt.Println("Error parsing proxy URL:", err)
       os.Exit(1)
   }

   client := &http.Client{
       Transport: &http.Transport{
           Proxy: http.ProxyURL(proxyURL),
       },
   }

   req, err := http.NewRequest("GET", target, nil)
   if err != nil {
       fmt.Println("Error creating HTTP request:", err)
       os.Exit(1)
   }

   req.Header.Add("Keep-Alive", "Hi, I am a Hop-By-Hop header")

   resp, err := client.Do(req)
   if err != nil {
       fmt.Println("Error sending HTTP request:", err)
       os.Exit(1)
   }
```

[14] https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.5.1

```
    defer resp.Body.Close()

    fmt.Println("Response from server:")
    fmt.Println(resp.Status)
}
```

Command:

```
go run request.go
```

Output (received on 7as.es host):

```
[Host] => 7as.es
[User-Agent] => Go-http-client/1.1
[Accept-Encoding] => gzip
[Connection] => close
[Keep-Alive] => Hi, I am a Hop-By-Hop header
```

The root cause of this issue appears to be in the following code path:

Affected File:
https://github.com/v2fly/v2ray-core/blob/.../common/protocol/http/headers.go

Affected Code:

```
func RemoveHopByHopHeaders(header http.Header) {
    // Strip hop-by-hop header based on RFC:
    // http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.5.1
    // https://www.mnot.net/blog/2011/07/11/what_proxies_must_do

    header.Del("Proxy-Connection")
    header.Del("Proxy-Authenticate")
    header.Del("Proxy-Authorization")
    header.Del("TE")
    header.Del("Trailers")
    header.Del("Transfer-Encoding")
    header.Del("Upgrade")

    connections := header.Get("Connection")
    header.Del("Connection")
    if connections == "" {
        return
    }
    for _, h := range strings.Split(connections, ",") {
        header.Del(strings.TrimSpace(h))
    }
}
```

It is recommended to implement RFC2616[15] correctly, hence removing Keep-Alive headers.

[15] https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.5.1

Hardening Recommendations

This report area provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

V2R-01-001 WP1: Possible DYLIB Injection on MacOS Client *(Medium)*

The MacOS V2Ray binary was found to be vulnerable to DYLIB Injection attacks[16] due to the absence of a __RESTRICT segment and a hardened runtime in the Mach-O file. An attacker with the ability to set environment variables could exploit this vulnerability to inject dynamic libraries into a legitimate V2Ray process. This injection could allow arbitrary code execution within the process context, leading to potential unauthorized access, data theft, or system compromise. Confirmation of this vulnerability requires compiling a DYLIB library and utilizing the DYLD_INSERT_LIBRARIES environment variable, as outlined in the following steps:

Step 1: Create a DYLIB Library to Inject

```
#include <stdio.h>
#include <syslog.h>
__attribute__((constructor))

static void myconstructor(int argc, const char **argv)
{
printf("[+] dylib constructor called from %s\n", argv[0]);
syslog(LOG_ERR, "[+] dylib constructor called from %s\n", argv[0]);
}
```

Step 2: Compile the dynamic library

```
$ gcc -dynamiclib libtest.c -o libtest.dylib
```

Step 3: Inject the DYLIB Library into the target application

```
$ DYLD_INSERT_LIBRARIES=libtest.dylib ./v2ray
```

Output:

```
[+] dylib constructor called from ./v2ray
A unified platform for anti-censorship.
Usage:
        v2ray <command> [arguments]
```

[16] https://attack.mitre.org/techniques/T1574/006/

This can be also confirmed by searching the desired string in the log stream.

Command

```
$ log stream --style syslog --predicate 'eventMessage CONTAINS[c] "constructor"'
```

Output:

```
Filtering the log data using "composedMessage CONTAINS[c] "constructor""
Timestamp                       (process)[PID]
2024-03-04 09:30:37.959581-0300  localhost v2ray[2573]: (libtest.dylib) [+] dylib
constructor called from ./v2ray
```

To mitigate DYLIB injection risks associated with the DYLD_INSERT_LIBRARIES environment variable on MacOS, a restricted segment should be enabled to prevent dynamic loading of *dylib* libraries for arbitrary code injection. It is recommended to use the following compiler options to enable the restricted segment feature:

Proposed fix 1 (compiler options):

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

Alternatively, a hardened runtime entitlement[17] could be set on the Mach-O binary:

Proposed fix 2 (hardened runtime entitlement):

```
codesign -s CERT --option=runtime v2ray
```

Command (check for hardened options)

```
codesign -dv ./v2ray
```

Output:

```
Executable=/Users/dani/Downloads/v2ray-macos-arm64-v8a/v2ray
Identifier=v2ray
Format=Mach-O thin (arm64)
CodeDirectory     v=20500     size=268422     flags=0x10000(runtime)     hashes=8383+2
location=embedded
Signature size=1644
Signed Time=7 Mar 2024 at 09:07:43
Info.plist=not bound
TeamIdentifier=not set
Runtime Version=11.0.0
Sealed Resources=none
Internal requirements count=1 size=84
```

[17] https://developer.apple.com/documentation/security/hardened_runtime

V2R-01-002 WP1: Potential MitM Attacks via TLS MinVersion *(Low)*

During the code review, it was discovered that the V2Ray codebase does not limit the TLS version to those considered safe at the time of writing. According to *RFC 8996*[18], TLS 1.0 and 1.1 have been officially deprecated since March 2021. Moreover, all major browsers have ceased support for these TLS versions due to known vulnerabilities[19]. The lack of a *MinVersion* setting in the TLS configurations is problematic, leading to clients defaulting to a minimum of TLS 1.2, while servers default to TLS 1.0. This discrepancy could allow a malicious attacker to conduct Man-In-The-Middle (MitM) attacks on V2Ray users. The issue originates from the following files:

Affected Files:
https://github.com/v2fly/v2ray-core/blob/…/transport/internet/tls/config.go#L198-L227
https://github.com/v2fly/v2ray-core/blob/…/app/dns/nameserver_quic.go#L380-L391
https://github.com/v2fly/v2ray-core/blob/…/infra/conf/cfgcommon/tlscfg/tls.go#L28-L62
https://github.com/v2fly/v2ray-core/blob/…/transport/internet/quic/dialer.go#L195-L201
https://github.com/v2fly/v2ray-core/blob/…/transport/internet/quic/hub.go#L88-L93

Example Code:
```
func (c *Config) GetTLSConfig(opts ...Option) *tls.Config {
[...]
config := &tls.Config{
    ClientSessionCache:     globalSessionCache,
    RootCAs:               root,
    InsecureSkipVerify:     c.AllowInsecure,
    NextProtos:            c.NextProtocol,
    SessionTicketsDisabled: !c.EnableSessionResumption,
    VerifyPeerCertificate:  c.verifyPeerCert,
    ClientCAs:              clientRoot,
}
```

Exceptions may occur when specific knowledge requires support for legacy clients, necessitating the use of older TLS versions for compatibility. However, from 2024, it is advised to set and enforce TLS1.3[20] as the minimum version, which is widely supported and available for nearly six years[21]. This is achieved by configuring TLS instances with the *MinVersion* setting to *tls.VersionTLS13*.

[18] https://datatracker.ietf.org/doc/rfc8996/
[19] https://www.zdnet.com/article/browsers-to-block-access-to-https-sites-using-tls-1-0-and-1-1-...
[20] https://www.vertexcybersecurity.com.au/tls1-2-end-of-life/
[21] https://www.internetsociety.org/blog/2018/08/internet-security-gets-a-boost/

V2R-01-003 WP1: Usage of Insecure PRNG *(Low)*

V2Ray was found to use the weak random number generator from the *math/rand* package instead of the more secure *crypto/rand*[22] alternative, lacking *Cryptographically-Secure Pseudorandom Number Generator (CSPRNG)*[23] capabilities. Usage of this suboptimal choice makes the security of the application more brittle and should be avoided. The issue is evident in the following files:

Affected Files:
https://github.com/v2fly/v2ray-core/blob/…/proxy/shadowsocks2022/encoding.go#L65
https://github.com/v2fly/v2ray-core/blob/…/proxy/shadowsocks/protocol.go#L106
https://github.com/v2fly/v2ray-core/blob/…/proxy/shadowsocks/protocol.go#L173
https://github.com/v2fly/v2ray-core/blob/…/proxy/shadowsocks/protocol.go#L192

Example Code:
```
import (
[...]
    "math/rand"
[...]
func (t *TCPRequest) EncodeTCPRequestHeader(effectivePsk []byte,
    eih [][]byte, address DestinationAddress, destPort int, initialPayload []byte, out
*buf.Buffer,
)
    [...]
    paddingLength := TCPMinPaddingLength
    if initialPayload == nil {
        initialPayload = []byte{}
        paddingLength += 1 + rand.Intn(TCPMaxPaddingLength) // TODO INSECURE RANDOM
USED
    }
```

It is advised to substitute the *math/rand* package with *crypto/rand*, which generates random bytes via a *Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)*, offering enhanced randomness. Generally, deploying CSPRNGs is recommended unless reproducibility is explicitly required. Utilizing deterministic *Pseudo-Random Number Generators (PRNGs)* poses the risk of unintended issues, highlighting the need for careful consideration.

[22] https://pkg.go.dev/crypto/rand
[23] https://en.wikipedia.org/wiki/Cryptographically-secure_pseudorandom_number_generator

V2R-01-004 WP1: Incorrect Default File Permissions *(Low)*

V2Ray was found to create files with permissions beyond "0o600", granting read and write access outside the intended user/owner[24]. This exceeds recommended practices, contravening the security principle of least privilege[25], which dictates using files with the minimum permissions necessary for operations. An attacker might exploit this to access configuration files in specific scenarios like shared hosting. Enforcing strict file permissions is crucial for protecting against vulnerabilities, including information disclosure and unauthorized code execution.

Affected Files:
https://github.com/v2fly/v2ray-core/blob/…/crypto/internal/chacha_core_gen.go#L59
https://github.com/v2fly/v2ray-core/blob/…/common/errors/errorgen/main.go#L20
https://github.com/v2fly/v2ray-core/blob/…/common/platform/filesystem/file.go#L53
https://github.com/v2fly/v2ray-core/blob/…/infra/vprotogen/main.go#L283

Example Code:

```
func CopyFile(dst string, src string) error {
    bytes, err := ReadFile(src)
    if err != nil {
        return err
    }
    f, err := os.OpenFile(dst, os.O_CREATE|os.O_WRONLY, 0o644)
    if err != nil {
        return err
    }
    defer f.Close()

    _, err = f.Write(bytes)
    return err
}
```

It is recommended to enforce access restrictions for files with confidential information to the owning user or service, and to the designated group only if needed, thereby eliminating global or external access. This measure aims to boost system resilience and protect sensitive data during program execution.

[24] https://security.openstack.org/guidelines/dg_apply-restrictive-file-permissions.html
[25] https://en.wikipedia.org/wiki/Principle_of_least_privilege

V2R-01-005 WP1: Multiple Vulnerable Dependencies *(Low)*

Retest Notes: V2Ray fixed this issue and 7ASecurity confirmed that the fix is valid.

The V2Ray project utilizes components with known vulnerabilities from underlying dependencies. Although most vulnerabilities are likely not exploitable in the current implementation, this practice could lead to security vulnerabilities. Below is a summary table of the known vulnerabilities in packages used directly or as underlying dependencies in V2Ray:

Component	Issues	Severity
go:github.com/cloudflare/circl:v1.3.3	CIRCL Kyber cryptographic library is susceptible to a timing side-channel vulnerability, identified as "kyberslash2"[26].	High
go:golang.org/x/crypto:v0.16.0	The SSH transport protocol with certain OpenSSH extensions, found in OpenSSH before 9.6 and other products, allows remote attackers to bypass integrity[27].	High
go:github.com/quic-go/quic-go:v0.40.0	quic-go[28] is an implementation of the QUIC protocol (RFC 9000, RFC 9001, RFC 9002) in Go. An attacker can cause its peer to run out of memory sending a large number of "PATH_CHALLENGE" frames.	Medium

This issue was confirmed by reviewing the following file:

Affected File:
https://github.com/v2fly/v2ray-core/blob/…/go.sum

Affected Code:

```
require (
   github.com/aead/cmac v0.0.0-20160719120800-7af84192f0b1 // indirect
   github.com/ajg/form v1.5.1 // indirect
   github.com/andybalholm/brotli v1.0.5 // indirect
   github.com/boljen/go-bitmap v0.0.0-20151001105940-23cd2fb0ce7d // indirect
   github.com/bufbuild/protocompile v0.6.0 // indirect
   github.com/cloudflare/circl v1.3.3 // indirect
```

[26] https://devhub.checkmarx.com/cve-details/Cx3111c14e-80ff/
[27] https://devhub.checkmarx.com/cve-details/CVE-2023-48795/
[28] https://devhub.checkmarx.com/cve-details/CVE-2023-49295/

```
    github.com/davecgh/go-spew v1.1.1 // indirect
[...]
```

It is recommended to upgrade all underlying dependencies to their current versions to resolve the above issues. To avoid similar issues in the future, an automated task and/or commit hook should be created to regularly check for vulnerabilities in dependencies. Some solutions that could help in this area are the *yarn audit* command[29], the *Snyk* tool[30], and the *OWASP Dependency Check* project[31]. Ideally, such tools should be run regularly by an automated job that alerts a lead developer or administrator about known vulnerabilities in dependencies so that the patching process can start in a timely manner.

V2R-01-009 WP1: Possible DoS Attacks on HTTP Services *(Medium)*

Some V2Ray HTTP services use the *net/http* package serve function without timeout settings, or fail to set timeouts where possible. This oversight exposes the application to *Slowloris*[32] attacks, where attackers prolong connections by slowly sending data, risking *Denial-of-Service (DoS)* incidents. This issue is evident in the following code snippets:

Affected Files:
https://github.com/v2fly/v2ray-core/blob/…/roundtripper/httprt/httprt.go#L121-L126
https://github.com/v2fly/v2ray-core/blob/…/main/distro/debug/debug.go#L8-L10
https://github.com/v2fly/v2ray-core/blob/…/app/restfulapi/restful_api.go#L97-L102
https://github.com/v2fly/v2ray-core/blob/…/testing/servers/http/http.go#L29-L36
https://github.com/v2fly/v2ray-core/blob/…/app/browserforwarder/forwarder.go#L69-L97

Affected Code:

```
func (f *Forwarder) Start() error {
    if f.config.ListenAddr != "" {
        f.forwarder = handler.NewHttpHandle()
        f.httpserver = &http.Server{Handler: f}
        [...]
        go func() {
            if err := f.httpserver.Serve(listener); err != nil {
                newError("cannot serve http forward server").Base(err).WriteToLog()
            }
        }()
    }
    return nil
}
```

[29] https://classic.yarnpkg.com/lang/en/docs/cli/audit/
[30] https://snyk.io/
[31] https://owasp.org/www-project-dependency-check/
[32] https://www.imperva.com/learn/ddos/slowloris/

It is recommended to configure timeouts using a custom *http.Server* object with appropriate timeouts, instead of the *default http.ListenAndServe* and *http.Serve* functions that do not support timeout settings. The code below demonstrates how to correctly instantiate an *http.Server* object with set timeouts:

Proposed Fix:

```
func (f *Forwarder) Start() error {
    if f.config.ListenAddr != "" {
        f.forwarder = handler.NewHttpHandle()
        f.httpserver = &http.Server{
            ReadHeaderTimeout: 15 * time.Second,
            ReadTimeout: 15 * time.Second,
            WriteTimeout: 10 * time.Second,
            IdleTimeout: 30 * time.Second,
            Handler: f
        }
        [...]
```

V2R-01-010 WP1: General Binary Hardening Recommendations *(Info)*

Testing confirmed that the V2Ray Linux binaries do not leverage a number of compiler flags to mitigate potential memory-corruption vulnerabilities. As a result, the application remains unnecessarily prone to the associated risks.

Linux binaries fail to leverage the following memory corruption prevention flags:

- **Missing Stack canaries**: This defense mechanism is used to detect and prevent exploits from overwriting the return address.
- **Missing RELRO**: This leaves the GOT section writable. Without the RELRO flag, buffer overflows on a global variable can overwrite GOT entries.
- **Missing PIE:** The *Position Independent Executable (PIE)* flag is a security mechanism that enables *Address Space Layout Randomization (ASLR)*, which randomizes the location where system executables are loaded into memory.

Please note all the aforementioned findings can be confirmed using the *checksec.sh*[33] utility.

Command:

```
checksec.sh --file v2ray
```

Output:

```
RELRO           STACK CANARY   NX            PIE           RPATH   RUNPATH
No RELRO        No canary found   NX enabled    No PIE        No RPATH   No RUNPATH
```

It is recommended to compile all binaries using the *CGO_LDFLAGS='-fstack-protector'* command line argument[34]. Furthermore, incorporating *-ldflags "-s -w" -buildmode=pie* is advised, although it is noted that compatibility depends on the Linux distribution[35]. Utilizing these low-level build options adds an extra layer of security and further reduces the risk of memory corruption vulnerabilities.

[33] https://www.trapkit.de/tools/checksec/#releases
[34] https://github.com/docker-library/golang/issues/231#issuecomment-602054311
[35] https://github.com/docker-library/golang/issues/231#issuecomment-694788522

WP3: V2Ray Supply Chain Implementation

Retest Notes: The V2Ray team now archives the history of github releases and the action log indefinitely[36].

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[37], revealed a 742% average yearly increase in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[38], *Github*[39], *Magento*[40], *SolarWinds*[41], and *Codecov*[42], among many others. To mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[43], named *Supply-Chain Levels for Software Artifacts* (*SLSA*)[44].

This area of the report elaborates on the current state of the supply chain integrity implementation of the V2Ray project, as audited against the SLSA framework. SLSA assesses the security of software supply chains and aims to provide a consistent way to evaluate the security of software products and their dependencies.

The following sections elaborate on the results against versions 0.1 and 1.0 of the SLSA standard. At the time of this assignment, V2Ray components are hosted on public GitHub repositories.

The V2Ray project uses the GitHub public repository for distribution, with well-defined dependencies. Using a scripted build process, V2Ray improves consistency and speed with every release. These processes align closely with the principles of SLSA, notably enhancing Provenance. This signifies that not only is the build process documented, but also, the resulting artifacts are intricately linked to a known and controlled build environment.

While auditing the supply chain implementation, the V2Ray project provided several positive impressions that must be acknowledged here:

- The project provides clear and comprehensive documentation on how to set up and use V2Ray.

[36] https://www.v2fly.org/en_US/developer/intro/releasearchive.html
[37] https://www.sonatype.com/press-releases/2022-software-supply-chain-report
[38] https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
[39] https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
[40] https://sansec.io/research/rekoobe-fishpig-magento
[41] https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
[42] https://blog.gitguardian.com/codecov-supply-chain-breach/
[43] https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
[44] https://slsa.dev/spec/

- The application is Signed in OpenBSD Signify format, allowing the verification of binaries.
- With GitHub Actions, the release build process is automated, decreasing the possibility of human error and resulting in more dependable releases.
- The build is designed to be reproducible, allowing independent verification of the build process.
- The build process and its logs are public, allowing anyone to verify the process.

SLSA v1.0 Analysis and Recommendations

SLSA v1.0 defines a set of four levels that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **Build L0: No guarantees** represents the lack of SLSA[45].
- **Build L1: Provenance exists**. The package has provenance showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge[46].
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance[47].
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection[48].

To produce artifacts with a specific SLSA level, the responsibility is split between the *Build* platform and the *Producer*. Broadly speaking, the *Build* platform must strengthen the security controls to achieve a specific level, while the *Producer* must choose and adopt a *Build* platform capable of achieving a desired SLSA level, implementing security controls as specified by the chosen platform.

The following sections summarize the results of the software supply chain security implementation audit, based on the SLSA v1.0 framework. Green check marks indicate that evidence of the SLSA requirement was found.

Producer

A package producer is the organization that owns and releases the software. It might be an open-source project, a company, a team within a company, or even an individual. The producer must select a build platform capable of reaching the desired SLSA Build Level.

In the context of the SLSA v1.0 framework, V2Ray adherence to Build Level 3 (L3) is demonstrated through its management of software artifact provenance. The V2Ray project is hosted on GitHub, a platform capable of producing Build Level 3 provenance.

[45] https://slsa.dev/spec/v1.0/levels#build-l0
[46] https://slsa.dev/spec/v1.0/levels#build-l1
[47] https://slsa.dev/spec/v1.0/levels#build-l2
[48] https://slsa.dev/spec/v1.0/levels#build-l3

This ensures that V2Ray fulfills the requirement to "*Choose an appropriate build platform*." Moreover, the V2Ray artifact generation process is clearly defined. Each step is meticulously scripted, and the use of GitHub Actions is leveraged to produce builds for each supported platform, in this case, V2Ray meets the Producer requirements, specifically "*Follow a consistent build process*."

Furthermore, V2Ray ensures that provenance information is not only present but also effectively distributed through the respective package manager ecosystems. By leveraging these established ecosystems, V2Ray effectively satisfies the Producer requirements essential for achieving Build Level 3 (L3) within the SLSA framework.

Requirement	L1	L2	L3
Choose an appropriate build platform	✅	✅	✅
Follow a consistent build process	✅	✅	✅
Distribute provenance	✅	✅	✅

Build platform

A package build platform is the infrastructure used to transform the software from source to package. This includes the transitive closure of all hardware, software, persons, and organizations that may influence the build. A build platform is often a hosted, multi-tenant build service, but it could be a system of multiple independent rebuilders, a special-purpose build platform used by a single software project, or even the workstation of an individual.

The build process, scripted via GitHub Actions[49], meticulously produces authenticated provenance, adhering to the exits and authentic requirements of SLSA 1.0. With each execution, the build platform generates signed logs, ensuring the validity and structured provenance that meet the unforgettable requirement. This meticulous approach empowers consumers to validate authenticity through guaranteed integrity and defined trust levels, thereby achieving compliance with both Level 1 and Level 2 requirements of SLSA v1.0. Additionally, each build step was carried out via a hosted build platform, which ensured that the processes were carried out in a separate environment free from unintentional outside interference. This satisfies the SLSA v1.0 *Hosted* and *Isolated* requirements.

[49] https://github.com/v2fly/V2Ray-core/blob/master/.github/workflows/release.yml

Requirement	Degree	L1	L2	L3
Provenance generation	Exists	✅	✅	✅
	Authentic		✅	✅
	Unforgeable			✅
Isolation strength	Hosted		✅	✅
	Isolated			✅

In conclusion, the V2Ray project fully complies with SLSA v1.0 Level 3 standards, representing the best practices in the industry in terms of security, integrity, and reliability.

SLSA v0.1 Analysis and Recommendations

SLSA v0.1 defines a set of five levels[50] that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the noted requirement was found.

Source code control requirements:

Requirement	L1	L2	L3	L4
Version controlled	✅	✅	✅	✅
Verified history			✅	✅

[50] https://slsa.dev/spec/v0.1/levels

Retained indefinitely			⛔ (18 mo.)	⛔
Two-person reviewed				⛔

Build process requirements:

Requirement	L1	L2	L3	L4
Scripted build	✅	✅	✅	✅
Build service		✅	✅	✅
Build as code			✅	✅
Ephemeral environment			✅	✅
Isolated			✅	✅
Parameterless				✅
Hermetic				✅
Reproducible				✅

Common requirements:

This includes common requirements for every trusted system involved in the supply chain, such as source, build, distribution, etc:

Requirement	L1	L2	L3	L4
Security				⛔
Access				⛔
Superusers				⛔

Provenance requirements:

Requirement	L1	L2	L3	L4
Available	✅	✅	✅	✅
Authenticated		✅	✅	✅
Service generated		✅	✅	✅

Non-falsifiable			✅	✅
Dependencies complete				✅

Provenance content requirements:

Requirement	L1	L2	L3	L4
Identifies artifact	✅	✅	✅	✅
Identifies builder	✅	✅	✅	✅
Identifies build instructions	✅	✅	✅	✅
Identifies source code		✅	✅	✅
Identifies entry point			✅	✅
Includes all build parameters			✅	✅
Includes all transitive dependencies				✅
Includes reproducible info				✅
Includes metadata	✅	✅	✅	✅

In conclusion, although V2Ray is still not SLSA v0.1 L3 compliant, due to the available GitHub tools it is possible to reach level SLSA v0.1 L3 and L4 as follows:

- *A GitHub Artifact retention policy*[51] ought to be implemented to comply with the retained indefinitely requirements.
- *GitHub branch protection rules*[52] should be implemented to comply with the Two-person reviewed requirements.

[51] https://docs.github.com/en/organizations/managing-organization-settings/configuring-the-retention-[...]
[52] https://github.blog/2018-03-23-require-multiple-reviewers/

Conclusion

The *V2Ray* project defended itself well against a broad range of attack vectors. In fact, not a single critical or high severity issue could be identified during this engagement. Continued cycles of security testing and hardening will further fortify the platform, making it even more resistant to potential attacks.

7ASecurity would like to highlight several positive aspects of V2Ray, as observed by the audit team:

- The evaluation of V2Ray backend components revealed a high degree of resilience against common web application security threats. Specifically, no vulnerabilities were detected in areas such as Command Injection, SQL Injection (SQLi), Cross-Site Request Forgery (CSRF), Local File Inclusion (LFI), or Remote Code Execution (RCE) during the assessment.
- The source code is meticulously organized and documented, which facilitates the process of understanding its functionality.
- The configuration process for V2Ray, applicable to both client and server modes, is straightforward.
- The application demonstrates robustness against malformed request headers and stress scenarios.
- The presence of an extensive test suite underscores the application maintainability and adherence to best practices in writing modular and testable code.
- The consistent efforts in maintenance and updates by the development team reflect a strong dedication to ensuring security and fostering ongoing enhancement.
- A commitment to secure coding practices, particularly those pertinent to the Go programming language, is evident.
- The application support for diverse file formats in its release distributions, coupled with a uniform strategy for generating binaries with security checks, demonstrates a methodical approach to building and distribution.

The security of the V2Ray project components may be enhanced with a focus on the following areas:

- **Anti-Fingerprinting Improvements:** It is possible to prevent multiple fingerprinting scenarios by adding random User-Agent headers (V2R-01-006), randomized JA3 fingerprints (V2R-01-007), and improving adherence to RFCs (V2R-01-008).
- **Build Hardening:** MacOS (V2R-01-001) and Linux (V2R-01-010) binaries may be hardened against environment variable attacks and memory corruption vulnerabilities by leveraging a number of platform mechanisms.
- **Software Patching:** All V2Ray components should adhere to appropriate

software patching procedures, consistently applying security patches in a timely manner (V2R-01-005). In a day and age when a significant portion of code comes from underlying software dependencies, routine patching is crucial to prevent potential security vulnerabilities. Possible automation for this could include tools like *Snyk.io*[53] or *Renovate Bot*[54].

- **File Permissions**: Application files should have the minimum possible permissions for the V2Ray clients and servers to work, as this will limit the potential for privilege escalation and leaks (V2R-01-004).
- **Mitigation of possible DoS Attacks:** Adequate timeouts should be in place to eliminate the potential for certain types of DoS attacks (V2R-01-009).
- **Usage of Sound Cryptography:** V2Ray should reduce the potential for usage of insecure TLS protocols (V2R-01-002), and insecure *Pseudo-Random-Number-Generators (PRNG)* (V2R-01-003).
- **Supply Chain Hardening:** V2Ray could take advantage of a number of features like *GitHub Branch protection* rules and *GitHub Actions* to easily improve its Supply Chain security posture against the SLSA standard (WP3). A good starting point in this regard, could be to integrate automated tools like *slsa-github-generator*[55] and *slsa-verifier*[56] into the build process.

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This approach will not only significantly enhance the security posture of the platform but also contribute to a reduction in the number of tickets in future audits.

Once all recommendations in this report are addressed and verified, a more thorough review, ideally including another code audit, is highly recommended to ensure adequate security coverage of the platform. Future audits should ideally allocate a greater budget, enabling test teams to delve into more complex attack scenarios.

It is suggested to test the application regularly, at least once a year or when substantial changes are deployed, to make sure new features do not introduce undesired security vulnerabilities. Consistently following this approach will lead to a reduction in the number of security issues and fortify the application against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank the V2Ray team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the Open Technology Fund (OTF) for sponsoring this project.

[53] https://snyk.io/
[54] https://github.com/renovatebot/renovate
[55] https://github.com/slsa-framework/slsa-github-generator
[56] https://github.com/slsa-framework/slsa-verifier