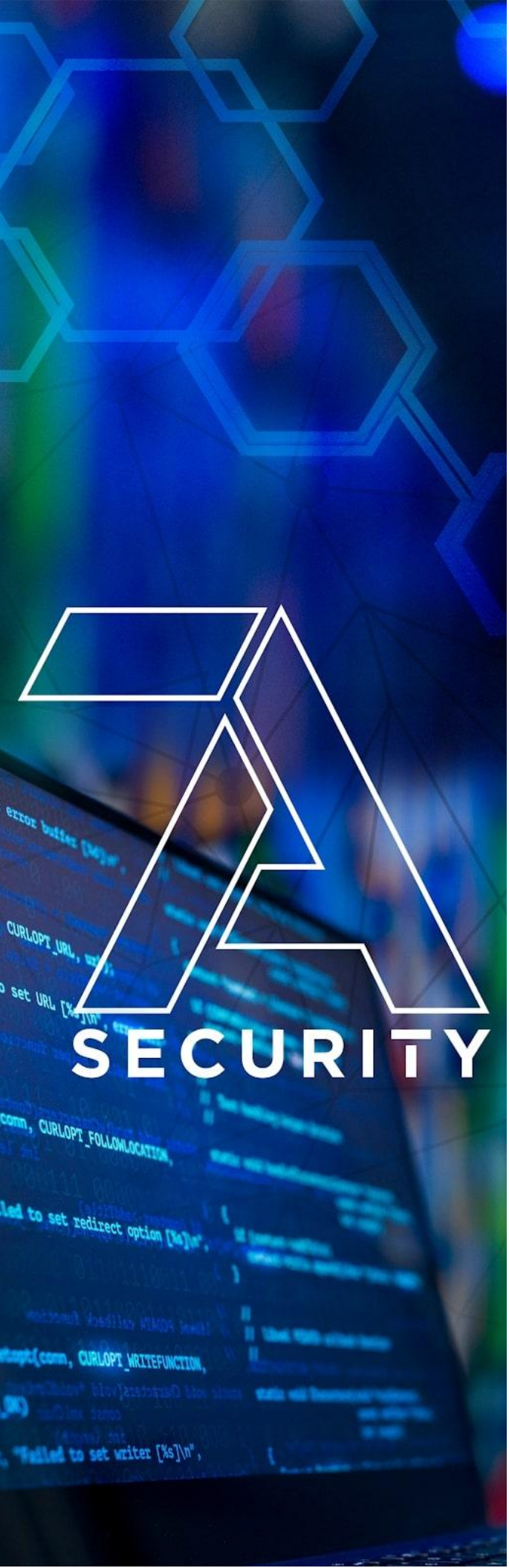


Test Target:
Psiphon Enhancements

# Pentest Report

---

## Client: Psiphon Inc.

**7ASecurity Test Team:**

- Abraham Aranguren, MSc
- Miroslav Stampar, MSc
- Patrick Ventuzelo, MSc

7ASecurity LLLP

*Protect Your Site & Apps From Attackers*

sales@7asecurity.com

# INDEX

**Introduction** **3**

**Identified Vulnerabilities** **5**

PSI-01-005 E2: Possible Client DoS via Traffic Tampering (Low) 5

**Miscellaneous issues** **8**

PSI-01-001 E2: Possible DoS via slice bounds out-of-range Weaknesses (Low) 8

PSI-01-002 E4: Possible DoS via nil pointer dereference (Info) 13

PSI-01-003 E4: Possible DoS via interface conversion panic (Info) 15

PSI-01-004 OOS: Windows Client DoS via known Temporary File Creation (Low) 16

**Conclusion** **19**

# Introduction

*"Psiphon is a circumvention tool from Psiphon Inc. that utilizes VPN, SSH and HTTP Proxy technology to provide you with uncensored access to Internet content. Your Psiphon client will automatically learn about new access points to maximize your chances of bypassing censorship."*

From https://www.psiphon3.com/en/index.html

This report documents the results of a security audit and best-practices review conducted by 7ASecurity, focusing on a number of Psiphon circumvention enhancements, including the code and components surrounding such features.

Given the nature of Psiphon, the 7ASecurity team performed this assignment assuming the role of government-sponsored adversaries with *Man-in-the-middle (MitM)* capabilities, aiming to defeat the censorship eluding features of Psiphon. Due to the public status of this report, the team will use less descriptive terms like *enhancement1*, *enhancement2*, *enhancement3* and *enhancement4* as well as *tactic1* and *tactic2, etc.* to prevent potential adversaries from gaining insight about Psiphon internals.

Psiphon requested 7ASecurity to perform a secure code audit on the following four recent circumvention-related enhancements:

- E1: *enhancement1*
- E2: *enhancement2*
- E3: *enhancement3*
- E4: *enhancement4*

The goal was to determine Psiphon's adherence to secure coding best practices and to provide safeguard recommendations where applicable.

During the test, a server configuration enabled with the aforementioned enhancements was facilitated to the 7ASecurity team by Psiphon. This ensured the team could fully focus on the security aspects during dynamic analysis of these new features at runtime without wasting any time on setup.

7ASecurity employed a whitebox methodology, which means the complete source code was available to the test team. Additionally, the Psiphon team provided commit links to all relevant changes for these circumvention enhancement features, which were reviewed thoroughly for adherence to security best practices. 7ASecurity used a combination of manual and automated tools for the code audit, performed code-level fuzzing on Psiphon source code as well as underlying third party libraries and

components, and network-level fuzzing using custom python scripts against both the Psiphon server and client components. The team further analyzed the network traffic between clients and servers. This guided the identification of potential attack vectors, fingerprinting potential and the understanding of how the Psiphon server and its clients switch from one tactic to another while they are under attack or when network communications fail. Lastly, the 7ASecurity team examined multiple public academic papers related to the new Psiphon circumvention enhancements to inform the fuzzing and attack scenarios prior to testing.

The test was conducted by 7ASecurity in April 2021 and yielded five security-specific discoveries. The security audit took a total of 18 working days and comprised three senior 7ASecurity penetration testers. The Psiphon platform was found to be resilient to a broad range of attack vectors and provided an overall solid impression. The system will notably be more difficult to attack once the issues in this report are resolved.

In the following sections, the report will elaborate on each finding and then offer a conclusion. Each finding has been discussed individually with its technical background and mitigation options. 7ASecurity delivers a detailed conclusion in the closing section. In light of the findings, the testing team comments on the security posture of the tested Psiphon framework and circumvention enhancements manifested during this assignment.

# Identified Vulnerabilities

The following sections list both vulnerabilities and implementation flaws identified during the testing period. Please note that the findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability has additionally been given a unique identifier (e.g. *PSI-01-001*) for the purpose of facilitating any future follow-up correspondence.

## PSI-01-005 E2: Possible Client DoS via Traffic Tampering (*Low*)

While reviewing the circumvention enhancements related to *enhancement2*, the 7ASecurity team conducted thorough network traffic fuzzing attacks against both the server and client components of Psiphon. In all cases, only the tested component was inside a virtual environment while the traffic being sniffed and fuzzed passed through the virtual network, making it easy to capture and manipulate. As a result, the team achieved partial *DoS* success against the client component, manifested with forced reconnects, failed proxy client download attempts, and unstable program behavior in rare attempts. As these findings were difficult to replicate and Psiphon showed a great ability to recover from these in most cases, the severity of this issue was set to *Low*. A malicious attacker could attempt to leverage these weaknesses to initiate *DoS* attacks against Psiphon clients hence disrupting Psiphon users, especially when they download large files.

**Issue 1: DoS via tactic1 redacted packets**

In order to test the Psiphon circumvention enhancements for *enhancement2*, *tactic1 redacted* attacks were emulated as used by the *Great Firewall of China*[1], where three specially crafted *redacted* packets were sent, trying to bring down the established connection between the client and third-party server component (i.e. sponsor). Web browser downloads proxied through the Psiphon client could be consistently stopped in this fashion:

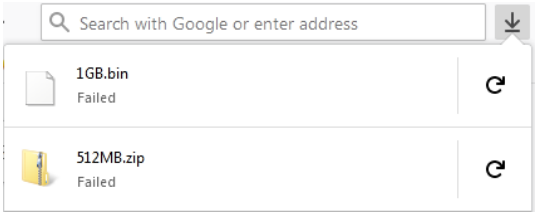


*Fig.: Result of the successful redacted attack*

[1] https://www.cl.cam.ac.uk/~rnc1/ignoring.pdf

Consequently, Psiphon clients, in an attempt to mitigate the detected communication problems, either reconnected or switched to alternative tactics.

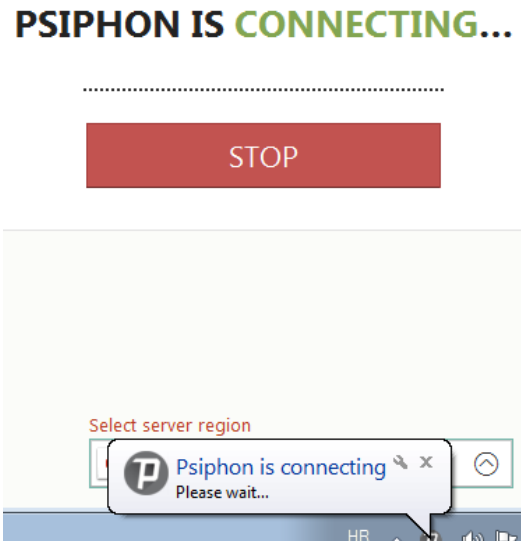


*Fig.: Reconnecting of Psiphon client after a successful redacted attack*

This area was tested using the following *PoC* python script:

**redacted attack PoC:**

```
Redacted, shared with Psiphon Team
```

Following the *redacted* attacks, a slightly modified version of the presented script was run, where a fake randomly generated (i.e. fuzzed) body was injected into the first *reply* packet. The main goal was to disrupt the Psiphon *tactic1* client-server communication with a minimum number of packets, while forcing more time consuming client reconnects or some other unstable program behavior.

This attack would be particularly disruptive to users downloading large files or videos, as the download will be interrupted and render the downloaded file unusable. However, Psiphon clients will eventually workaround *tactic1* attacks if they switch to *tactic2* tactics.

**Issue 2: DoS via tactic2 traffic**

Several *tactic2* attacks were attempted during this assignment. As the goal was to disrupt *tactic2* communications, multiple strategies were implemented, where each one was attempted individually, as well as concurrently. These were called *replay*, *fuzz*, *random*, *empty*, *big* and *reflect* by the test team and are briefly explained below:

| Strategy | Description |
|---|---|
| *replay* | The last packet coming from one side was stored and replayed when a subsequent packet coming from the other side was recognized (i.e. where the client expected a server response). |
| *fuzz* | The same mechanism was used in the *replay* strategy, with the exception of the *tactic2* body being modified at random locations with randomly chosen bytes. |
| *random* | Everything was the same as in the *fuzz* strategy, with the exception of the *tactic2* body being filled with randomly chosen content. |
| *empty* | The *tactic2* packet body was blank. |
| *big* | The *tactic2* packet body was filled with large randomly chosen content. |
| *reflect* | Source and destination IP addresses were exchanged within the captured *tactic2* packet, and finally sent to the source. |

The example below corresponds to one of the *fuzz* strategies attempted during testing:

**Replay Fuzz PoC**:

```
Redacted, shared with Psiphon Team
```

Although the team managed to create a number of forced reconnect events, it took more than 20 minutes of parallel attacks. Furthermore, in one case, a continuous high *CPU* usage event occurred while there was no active traffic being proxied. Nevertheless, such attack vectors seemed impractical due to the extended time window required, in combination with the non-deterministic nature of the rare *DoS* events identified.

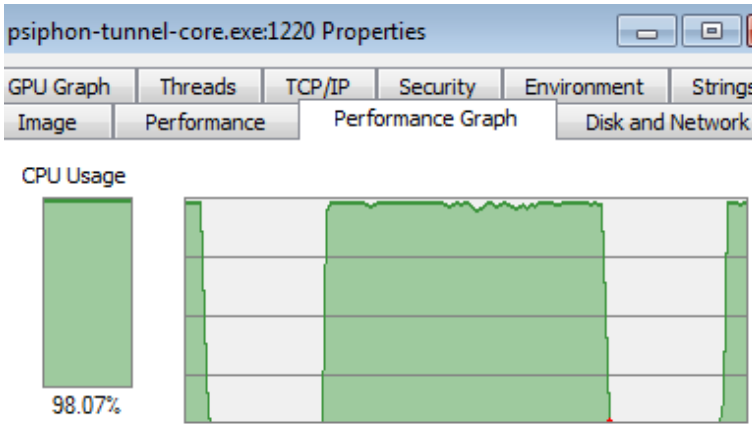


*Fig.: Unstable state of high CPU usage in one case of prolonged tactic2 fuzzing*

Please note, that while the 7ASecurity team was moderately successful in achieving *DoS* via *tactic1 redacted* attacks against the client component (described above), *tactic2* communication proved to be significantly more resilient. At least, the solution appeared to be robust against the utilized strategies.

It is recommended to consider possible improvements to enhance the reliability of connections without revealing more information than needed to adversaries. Some ideas to achieve this could be to switch to *tactic2* alternative tactics as fast as possible in case of subsequent problems with *tactic1* tactics. Another potential strategy could be to use two or more concurrent connections, for example, a *tactic1* connection and a *tactic2* connection. This could speed up the switch from one connection to another when a given connection fails or help avoid certain disruptions completely if attacks target only one of the two protocols (i.e. *tactic1* or *tactic2*). For additional research and continuous improvement of the resiliency of Psiphon, the 7ASecurity team shared all of the implemented *PoC* fuzzing scripts with the Psiphon Team.

**Retest Notes:** The Psiphon Team accepts the risk for this item for the time being, but will leverage this information and the provided test scripts for improving the resilience of all Psiphon clients in the future.

# Miscellaneous issues

This section covers notable findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are weaknesses that did not provide an easy way to be exploited. To conclude, while a vulnerability is present, an exploit might not always be possible.

## PSI-01-001 E2: Possible DoS via slice bounds out-of-range Weaknesses (*Low*)

While code auditing and fuzzing the Psiphon Tunnel source code, it was found that a number of functions currently fail to perform a data length check prior to slicing. This led to the following crash error: *"panic: runtime error: slice bounds out of range"*. This issue is considered to be of low severity for two reasons: First, Psiphon makes use of the *panicwrap* package[2]. Although this is only used to log panic exceptions, in practice it may also be used to restart the server in case of a crash, hence mitigating most *DoS* scenarios. Furthermore, it seems that this attack would only be exploitable by an attacker able to specify a malicious upstream *NTLM proxy* inside a configuration file. However, it was later discovered that the *ParseChallengeMessage* vulnerability described below has a slightly higher impact, as a malicious attacker could trigger this attack by fooling Psiphon client users to run a tampered configuration (i.e. through social engineering, trojaned versions of Psiphon or vulnerabilities in Psiphon clients or their platforms: Android, iOS, Windows, etc.). A malicious attacker able to manipulate authentication messages might leverage these weaknesses to slow down and eventually *DoS* a Psiphon server or a Psiphon client.

Please note, that these issues were identified via manual code review and were then confirmed using direct fuzzing of the affected functions, these are described below:

**Issue 1: Slice bounds out-of-range on *ParseChallengeMessage***

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/f1863f4f24bbdeb37d04767a0982adad7bedb956/psiphon/upstreamproxy/go-ntlm/ntlm/message_challenge.go#L56

**Affected Code:**

```
func ParseChallengeMessage(body []byte) (*ChallengeMessage, error) {
    challenge := new(ChallengeMessage)

    challenge.Signature = body[0:8]
    if !bytes.Equal(challenge.Signature, []byte("NTLMSSP\x00")) {
        return challenge, errors.New("Invalid NTLM message signature")
    }
[...]
```

[2] https://github.com/mitchellh/panicwrap

Please note that this function is used outside the *go-ntlm* package via **NTLMHttpAuthenticator*. This *NTLMHttpAuthenticator* structure, defined inside ***auth_ntlm.go*** *is then used inside* ***http_authenticator.go*** by *NewHttpAuthenticator*. This function is used multiple times and appears to be a core element of the *upstreamproxy* package, as illustrated by the evidence below:

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/8103eb8f978b15757852a5fcf24d2361db890a92/psiphon/upstreamproxy/proxy_http.go#L196

**Affected Code:**

```
pc.authenticator, authErr = NewHttpAuthenticator(resp, username, password)
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/8103eb8f978b15757852a5fcf24d2361db890a92/psiphon/upstreamproxy/transport_proxy_auth.go#L132

**Affected Code:**

```
authenticator, err := NewHttpAuthenticator(
```

**Issue 2: Slice bounds out-of-range on *ParseAuthenticateMessage***

**Affected File:**
*https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/f1863f4f24bbdeb37d04767a0982adad7bedb956/psiphon/upstreamproxy/go-ntlm/ntlm/message_authenticate.go#L56*

**Affected Code:**

```
func ParseAuthenticateMessage(body []byte, ntlmVersion int) (*AuthenticateMessage,
error) {
    am := new(AuthenticateMessage)

    am.Signature = body[0:8]
    if !bytes.Equal(am.Signature, []byte("NTLMSSP\x00")) {
        return nil, errors.New("Invalid NTLM message signature")
    }
[...]
```

Please note similar issues exist within the same file:

**Affected Code:**

```
59:     am.Signature = body[0:8]
```

```
64:     am.MessageType = binary.LittleEndian.Uint32(body[8:12])
127:            am.NegotiateFlags = binary.LittleEndian.Uint32(body[offset : offset+4])
132:                    am.Version, err = ReadVersionStruct(body[offset : offset+8])
148:                    am.Mic = body[offset : offset+16]
153:    am.Payload = body[offset:]
```

The 7ASecurity team confirmed that the vulnerable function appears to be used only inside the *go-ntlm* package.

**Issue 3: Multiple slice bounds out-of-range on *psiphon/upstreamproxy***

Similar issues were found in the *upstreamproxy* package.

**Affected Directory:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/tree/7d4307b1a387df94c8410ecdf20223b3db8eab7f/psiphon/upstreamproxy

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/7d4307b1a387df94c8410ecdf20223b3db8eab7f/psiphon/upstreamproxy/go-ntlm/ntlm/av_pairs.go#L134

**Affected Code:**

```go
func ReadAvPair(data []byte, offset int) *AvPair {
    pair := new(AvPair)
    pair.AvId = AvPairType(binary.LittleEndian.Uint16(data[offset : offset+2]))
    pair.AvLen = binary.LittleEndian.Uint16(data[offset+2 : offset+4])
    pair.Value = data[offset+4 : offset+4+int(pair.AvLen)]
    return pair
}
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/7d4307b1a387df94c8410ecdf20223b3db8eab7f/psiphon/upstreamproxy/go-ntlm/ntlm/keys.go#L9

**Affected Code:**

```
9:      part1, err = des(lmnowf[0:7], lmChallengeResponse[0:8])
15:     part2, err = des(key, lmChallengeResponse[0:8])
22:     keyExchangeKey = concat(lmnowf[0:8], zeroBytes(8))
50:             sealKey = randomSessionKey[0:7]
52:             sealKey = randomSessionKey[0:5]
61:             sealKey = concat(randomSessionKey[0:7], []byte{0xA0})
63:             sealKey = concat(randomSessionKey[0:5], []byte{0xE5, 0x38, 0xB0})
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/7d4307b1a387df94c8410ecdf20223b3db8eab7f/psiphon/upstreamproxy/go-ntlm/ntlm/challenge_responses.go#L25

**Affected Code:**

```
25:     r.Response = bytes[0:24]
88:     r.Response = bytes[0:16]
102:    c.TimeStamp = bytes[24:32]
103:    c.ChallengeFromClient = bytes[32:40]
106:    c.AvPairs = ReadAvPairs(bytes[44:])
119:    r.Response = bytes[0:24]
141:    r.Response = bytes[0:16]
142:    r.ChallengeFromClient = bytes[16:24]
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/621f90ee7ddf845eee150e8acfbe246bc5a8c627/psiphon/upstreamproxy/go-ntlm/ntlm/version.go#L19

**Affected Code:**

```
24:     versionStruct.ProductBuild = binary.LittleEndian.Uint16(structSource[2:4])
25:     versionStruct.Reserved = structSource[4:7]
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/621f90ee7ddf845eee150e8acfbe246bc5a8c627/psiphon/upstreamproxy/go-ntlm/ntlm/message_challenge.go#L56

**Affected Code:**

```
59:     challenge.Signature = body[0:8]
64:     challenge.MessageType = binary.LittleEndian.Uint32(body[8:12])
76:     challenge.NegotiateFlags = binary.LittleEndian.Uint32(body[20:24])
78:     challenge.ServerChallenge = body[24:32]
80:     challenge.Reserved = body[32:40]
92:             challenge.Version, err = ReadVersionStruct(body[offset : offset+8])
99:     challenge.Payload = body[offset:]
```

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/621f90ee7ddf845eee150e8acfbe246bc5a8c627/psiphon/upstreamproxy/go-ntlm/ntlm/payload.go#L80

**Affected Code:**

```
84:     p.Len = binary.LittleEndian.Uint16(bytes[startByte : startByte+2])
85:     p.MaxLen = binary.LittleEndian.Uint16(bytes[startByte+2 : startByte+4])
```

```
86:     p.Offset = binary.LittleEndian.Uint32(bytes[startByte+4 : startByte+8])
90:             p.Payload = bytes[p.Offset:endOffset]
```

In order to resolve these issues, it is recommended to perform appropriate data length checks prior to slicing.

**Retest Notes:** The Psiphon team promptly addressed the issue[3] and the fix was reviewed by 7ASecurity. The issue has been resolved.

### PSI-01-002 E4: Possible DoS via nil pointer dereference (*Info*)

While fuzzing the **psiphon/common/osl** package, it was found that the function ***LoadConfig*** fails to validate *config.scheme* fields, which led to the following crash error: *"panic: runtime error: invalid memory address or nil pointer dereference"*. This issue can be reproduced by providing a syntactically correct *JSON psiphond-osl.config* file that does not contain the expected *Scheme.Epoch* field. This issue is considered to be of *Info* severity for similar reasons as PSI-01-001. An attacker needs to supply or social engineer a psiphon server admin to run a crafted configuration file.

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/96544cdff9f54fecc93a539ba1e89535eb1d0111/psiphon/common/osl/osl.go#L284-L298

**Affected Code:**

```go
func LoadConfig(configJSON []byte) (*Config, error) {

    var config Config
    err := json.Unmarshal(configJSON, &config)
    if err != nil {
        return nil, errors.Trace(err)
    }

   var previousEpoch time.Time

   for _, scheme := range config.Schemes {
        epoch, err := time.Parse(time.RFC3339, scheme.Epoch)
        if err != nil {
               return nil, errors.Tracef("invalid epoch format: %s", err)
        }
```

[3] https://github.com/Psiphon-Labs/psiphon-tunnel-core/pull/599/commits/6d3ac...

The crash can be confirmed with a crafted configuration file as follows:

**Command:**

```
cat psiphond-osl.config
```

**Output:**

```
{"Schemes":[null]}
```

**Command:**

```
./psiphond run
```

**Output:**

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x86be6f]

goroutine 1 [running]:
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.LoadConfig(0xc00
0041200, 0x14, 0x600, 0x0, 0x434266, 0x0)

/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/osl.go:2
97 +0x36f
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.NewConfig.func1(
0xc000041200, 0x14, 0x600, 0x34aec0fe, 0xed800c460, 0x15d07e0, 0x0, 0x0)

/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/osl.go:2
66 +0x4c
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common.(*ReloadableFile).Re
load(0xc00007e900, 0xc0001b7700, 0x0, 0x0)

/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/reloader.go:
179 +0x34a
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.NewConfig(0xc000
039f20, 0x13, 0xc000186700, 0x0, 0x0)

/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/osl.go:2
75 +0x146
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server.NewSupportServices(0
xc0001dae00, 0x0, 0x0, 0xc0001dae00)

/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server/services.go:
452 +0x94
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server.RunServices(0xc00012
6a00, 0x1118, 0x1318, 0x0, 0x0)
```

```
/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server/services.go:
74 +0x155
main.main()
       /go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/Server/main.go:259
+0x14c1
{"build_rev":"577a7b05","event_name":"server_panic","host_id":"example-host-id"
,"panic":"panic: runtime error: invalid memory address or nil pointer
dereference\n[signal SIGSEGV: segmentation violation code=0x1 addr=0x8
pc=0x86be6f]\n\ngoroutine 1
[running]:\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.Load
Config(0xc000041200, 0x14, 0x600, 0x0, 0x434266,
0x0)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/
osl.go:297
+0x36f\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.NewConfi
g.func1(0xc000041200, 0x14, 0x600, 0x34aec0fe, 0xed800c460, 0x15d07e0, 0x0,
0x0)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/
osl.go:266
+0x4c\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common.(*ReloadableF
ile).Reload(0xc00007e900, 0xc0001b7700, 0x0,
0x0)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/relo
ader.go:179
+0x34a\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl.NewConfi
g(0xc000039f20, 0x13, 0xc000186700, 0x0,
0x0)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/common/osl/
osl.go:275
+0x146\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server.NewSupportSe
rvices(0xc0001dae00, 0x0, 0x0,
0xc0001dae00)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/se
rver/services.go:452
+0x94\ngithub.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server.RunServices(0
xc000126a00, 0x1118, 0x1318, 0x0,
0x0)\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon/server/serv
ices.go:74
+0x155\nmain.main()\n\t/go/src/github.com/Psiphon-Labs/psiphon-tunnel-core/Serv
er/main.go:259 +0x14c1\n","timestamp":"2021-04-08T11:40:50+02:00"}
```


In order to resolve this issue, it is recommended to validate that *scheme* fields are not null before usage.

**Retest Notes:** The Psiphon team promptly addressed the issue[4] and the fix was reviewed by 7ASecurity. The issue has been resolved.

[4] https://github.com/Psiphon-Labs/psiphon-tunnel-core/commit/280b...

### PSI-01-003 E4: Possible DoS via interface conversion panic (*Info*)

During the fuzzing process of the ***psiphon/common*** package, it was found that the ***GetNotice*** function fails to perform a conversion of the payload as a generic map, which led to the following crash error: *"panic: interface conversion: interface {} is float64, not map[string]interface {}"*. Please note that the *GetNotice* function is only used for testing within the *psiphon-tunnel-core* source code, making this issue only affect projects using *psiphon-tunnel-core* as a third party library or copying the vulnerable code into another project.

**Affected File:**
https://github.com/Psiphon-Labs/psiphon-tunnel-core/blob/08f530bd796fe1b483480b8849de2d1c24476f27/psiphon/notice.go#L976

**Affected Code:**

```
func GetNotice(notice []byte) (
    noticeType string, payload map[string]interface{}, err error) {

    var object noticeObject
    err = json.Unmarshal(notice, &object)
    if err != nil {
        return "", nil, err
     }
     var objectPayload interface{}
     err = json.Unmarshal(object.Data, &objectPayload)
     if err != nil {
         return "", nil, err
     }
     return object.NoticeType, objectPayload.(map[string]interface{}), nil
}
```

This issue was verified using the following Proof-of-Concept (PoC) code:

**PoC code:**

```
func panic_interface_conversion_GetNotice() {
    data := []byte("{\"data\":5}")
    _, _, _ = psiphon.GetNotice(data)

}
```

**Output:**

```
panic: interface conversion: interface {} is float64, not map[string]interface
{}

goroutine 17 [running, locked to thread]:
github.com/Psiphon-Labs/psiphon-tunnel-core/psiphon.GetNotice(0x3f8ad90, 0xb,
0xb, 0x0, 0xc000116000, 0x0, 0xc000000600, 0xc000000688)

github.com/Psiphon-Labs/psiphon-tunnel-core@v0.0.0-00010101000000-000000000000/
psiphon/notice.go:989 +0x1f1
```

In order to resolve this issue, *objectPayload* should be verified after *json.Unmarshal*.

**Retest Notes:** The Psiphon team promptly addressed the issue[5] and the fix was reviewed by 7ASecurity. The issue has been resolved.

### PSI-01-004 OOS: Windows Client DoS via known Temporary File Creation (*Low*)

While looking for weaknesses in the implementation of *enhancement2*, it was discovered that the *Psiphon GUI* client for Windows (*psiphon3.exe)* uses a known temporary file location (*%TEMP%\psiphon-tunnel-core.exe)* for downloading the tunneling client, instead of a random one. Although the *%TEMP%* directory will not be writable to non-admin users on the same computer, certain attack vectors remain to exploit this weakness. For example, a malicious application on the same computer, a malicious script run via a phishing attack, or getting the user to follow some fake tutorial steps from an attacker-controlled website are some plausible ways to create a file in this location. A malicious attacker with the ability to write or fool a user to write files on *%TEMP%* could leverage this weakness to create a temporary file with an identical filename and the read-only flag set. As a result, the *GUI* client will continuously check for the availability of the tunneling client, while causing high *CPU* usage. This type of the vulnerability is well known and generally referred to as *insecure temporary file creation*[6789].

This issue illustrates a simple way to prevent *Windows GUI* clients from operating, while the root cause might be difficult to be detected by the end user. *The Windows GUI* user would be prevented from running the auxiliary tunneling client application, subsequently forcing their traffic to be transferred in an unprotected manner.

[5] https://github.com/Psiphon-Labs/psiphon-tunnel-core/commit/77fe...
[6] https://owasp.org/www-community/vulnerabilities/Insecure_Temporary_File
[7] https://cwe.mitre.org/data/definitions/378.html
[8] https://rules.sonarsource.com/python/type/Vulnerability/RSPEC-5445
[9] https://securiteam.com/windowsntfocus/5NP0H0AC0Q/

This issue can be trivially reproduced using the following commands:

**Commands:**

```
echo > %TEMP%\psiphon-tunnel-core.exe
attrib +r %TEMP%\psiphon-tunnel-core.exe
psiphon3.exe
```

**Result:**

High *CPU* consumption and complete inability to use the *Windows GUI* client.

**Root Cause Analysis**

This issue occurs due to the following code, which writes the executable in a predictable temporary file location:

**Affected File:**

https://github.com/Psiphon-Inc/psiphon-windows/blob/1888ec6851392bffbd804452d3f5172c5e208c27/src/utilities.cpp#L102-L130

**Affected Code:**

```
TCHAR filePath[MAX_PATH];
if (NULL == PathCombine(filePath, tempPath.c_str(), exeFilename))
{
     my_print(NOT_SENSITIVE, false, _T("ExtractExecutable - PathCombine failed (%d)"),
GetLastError());
     return false;
}

HANDLE tempFile = INVALID_HANDLE_VALUE;
bool attemptedTerminate = false;
while (true)
{
     tempFile = CreateFile(filePath, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
     if (tempFile == INVALID_HANDLE_VALUE)
    {
         int lastError = GetLastError();
         if (!attemptedTerminate && ERROR_SHARING_VIOLATION == lastError)
         {
         if (succeedIfExists)
         {
                    // The file must exist, and we can't write to it, most likely
because it is
```

```
                    // locked by a currently executing process. We can go ahead and
consider the
                    // file extracted.
                    // TODO: We should check that the file size and contents are the
same. If the file
                    // is different, it would be better to proceed with attempting to
extract the
                    // executable and even terminating any locking process -- for
example, the locking
                    // process may be a dangling child process left over from before a
client upgrade.
                    path = filePath;
                    return true;
```

To resolve this issue, the use of a randomly chosen location for temporary storage[10] of the tunneling client is recommended. This will ensure that the *Windows GUI* client is resilient against similar attacks even in situations where attackers have write permissions on the same folder.

**Retest Notes:** The Psiphon team promptly addressed the issue[11] and the fix was reviewed by 7ASecurity. The issue has been resolved.

[10] https://stackoverflow.com/a/28005931
[11] https://github.com/Psiphon-Inc/psiphon-windows/commit/8d1a8b7f3c....

# Conclusion

The Psiphon platform made a robust impression against a broad range of attack vectors. This reflects well on the team behind the solution. 7ASecurity detected only 1 security vulnerability of low severity. Hence, no significant security flaws could be identified during this assignment. The remaining 4 findings were classified as miscellaneous weaknesses and thus, not considered as vulnerabilities. This represents a surprisingly low number of security-relevant discoveries considering the large attack surface available in the scope. Part of this appears to be down to the platform chosen for this solution - *Go*[12]. This programming language has a solid security track record with out-of-the-box protection against multiple attack vectors. Additionally, the Psiphon codebase has been audited multiple times by several security firms over the years, which makes identifying security issues increasingly difficult.

It is important to note that despite a thorough review of code changes for all enhancements, surrounding code, components in scope and even underlying libraries, 7ASecurity was unable to identify any weakness relevant to *enhancement1* and *enhancement3*. While reviewing the source code and functionality surrounding *enhancement2* and *enhancement4*, some issues were spotted, but most of these had a low impact as it would be complicated for an attacker to implement them. The miscellaneous issues identified during this assignment were characterized as such due to the prerequisites required for a successful exploit (PSI-01-001, PSI-01-002, PSI-01-003, PSI-01-004). For example, PSI-01-002 and PSI-01-003 require the Psiphon server to parse invalid configuration files to trigger a panic in at least some of the described scenarios. PSI-01-001 requires receiving an incorrect NTLM message from a malicious NTLM proxy. Another weakness identified while reviewing the items in scope for this assessment was PSI-01-004. However, 7ASecurity determined upon further examination that this latter finding was not directly related to any of the enhancements and hence, was considered out of scope (OOS).

The code audit, unsurprisingly, delivered similar positive impressions. The 7ASecurity team scrutinized all relevant source code using a combination of manual and automated analysis through multiple *Go* auditing tools. During the review, the code was found to be well documented, clean, professional looking, and intact, like the lifework of competent developers. No issues were identified while reviewing the cryptographic primitives and, with few exceptions, nothing in the code looked like a regular vulnerability. The *go-ntlm* package[13] was an exception here as it looked like it was never audited before. This has

[12] https://golang.org/
[13] https://github.com/Psiphon-Labs/psiphon-tunnel-core/tree/.../psiphon/upstreamproxy/go-ntlm

been illustrated through the number of issues mentioned in PSI-01-001. 7ASecurity suggests that this package is diminishing the overall code quality of Psiphon, which is otherwise clean, documented and polished.

In addition to the comprehensive code audit, 7ASecurity performed code-level fuzzing on multiple functions within the Psiphon source code as well as third party software components in charge of parsing network packets. These targets were fuzzed extensively without the identification of any vulnerability or weakness.

As part of this assignment, 7ASecurity also performed network-level traffic analysis and fuzzing. While testing for *DoS*, the Psiphon server and clients showed resilience against *DoS* attacks. The team was unable to find any way to negatively impact the performance or availability of Psiphon servers at runtime. The team finally managed to find a way to disrupt *tactic1* communications for Psiphon clients (PSI-01-005). However, clients proved to be more difficult to disrupt when using *tactic2* strategies. Please note that all similar attempts made were unsuccessful against the server component. Other server *DoS* tests included *SYN* floods. In certain the tests, the *Mausezahn* tool[14] was used, with different options in each attempt (i.e. *tactic1* body content, fake source IP addresses, traffic speed, etc.). The Psiphon server defended itself effectively against all such attempted attacks.

In general, the Psiphon server and clients were well configured and protected against bugs that could put user censorship bypassing at risk. Psiphon offered strong resistance against issues like *IP* blocking, *DoS* attacks via network-level fuzzing, code-level fuzzing, server fingerprinting and Psiphon server list extraction, as well as many other attempted attack vectors. Overall, the security posture painted a positive picture.

During the audit, the 7ASecurity team shared a Slack channel with the Psiphon team, which was used to promptly report all issues identified regardless of their severity. Communications were fluent and the Psiphon team were helpful and diligent at assisting 7ASecurity in answering all queries in a timely manner.

Despite best efforts made to cover the Psiphon circumvention enhancements thoroughly, it is important to note that the time for this assessment was limited and insufficient to cover all complexities of the Psiphon codebase, servers and clients in their entirety. Hence, it should be assumed that additional vulnerabilities still exist. It is recommended that the Psiphon team search for the security anti-patterns recognized in this assignment and attempt to identify additional areas with similar issues in the codebase. This should

[14] https://man7.org/linux/man-pages/man8/mausezahn.8.html

ideally be done before any other security audit. Future security audits are encouraged to be conducted in a similar whitebox fashion, whereby the audit team has access to source code. This will keep ensuring a comprehensive analysis and better fix suggestions, referencing specific lines of code, as could be done in this assignment.

It is recommended that all issues identified in this report, including informational and low severity tickets, are addressed where possible. This will not just strengthen the security posture of the platform, but also reduce the number of tickets in future security engagements.

In addition, it is advised that the platform continues to be tested regularly, at least once a year or in the event of substantial changes, before they are deployed, to ensure that new features do not introduce undesired security vulnerabilities. This proven strategy will continue to lower the number of security issues and make the Psiphon server and client components more resilient against online attacks over time.

Finally, 7ASecurity would like to thank Jessica Wever, Mike Fallone, Irv Simpson, Rod Hynes and the rest of the Psiphon Team for their excellent project coordination, admirable support and assistance, both before and during this assignment.