Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report Psiphon Tunnel Core Codebase 05.2024

Cure53, Dr.-Ing. M. Heiderich, M. Wege, Dr. A. Pirker, MSc. D. Weißer

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

> *"Psiphon Inc. is a company based in Toronto, producing open-source multi-platform software that helps over 3 million people every week connect to content on the Internet. We're a team focused on delivering the best software we can, introducing new products regularly and making sure we develop to the needs of our constantly growing global audience."*
>
> From https://www.psiphon.ca/en/about.html

This report describes the results of a security assessment of the Psiphon complex, with the focus on the Psiphon Tunnel Core codebase. The project, which included a penetration test and a dedicated source code audit, was carried out by Cure53 in the spring of 2024.

Registered as *PSI-08*, the examination was requested by Psiphon Inc. in January 2024 and then scheduled to start in late April 2024. This means both parties had ample time to prepare, although best practices have already been in place given prior cooperation.

In fact, this audit marks the eighth collaboration between Psiphon and Cure53. While it is worth clarifying that the Psiphon Tunnel Core code was part of the scope during the previous examination, this was the case during *PSI-01,* implemented back in June and July 2017. As such, quite a lot of time has passed since the last review.

In terms of the exact timeline and specific resources allocated to *PSI-08,* Cure53 completed the research from CW19 to CW21, which specifically signifies April and May in 2024. In order to achieve the expected coverage for this task, a total of thirty-two days were invested and a team of four senior testers was formed and assigned to the preparations, execution, documentation and delivery of this project.

Given the scope of this *PSI-08* project, the assessment was set to contain just one work package (WP):

- **WP1**: White-box pentests & audits against Psiphon Tunnel Core codebase

As the title of the WP indicates, white-box methodology was utilized. Cure53 was provided with binaries, as well as all further means of access required to complete the tests, the methodology chosen here was white-box. Additionally, all sources corresponding to the test targets were shared to make sure the project can be executed in line with the agreed-upon framework.

The project could be completed without any major problems. To facilitate a smooth transition into the testing phase, all preparations were completed in CW18, i.e., in the week prior to the technical examinations. Throughout the engagement, communications were conducted via a private, dedicated and shared Slack channel. Stakeholders - including the Cure53 testers and the internal staff from Psiphon Inc. - could participate in discussions in this space.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Not many questions had to be posed by Cure53 and the quality of all project-related interactions was consistently excellent. Ongoing exchanges contributed positively to the overall outcomes of this project. Significant roadblocks could be avoided thanks to clear and diligent preparation of the scope.

Cure53 offered frequent status updates about the test and the emerging findings. Selected discoveries were live-reported in the form of Markdown files, as requested by Psiphon.

The Cure53 team succeeded in achieving very good coverage of the WP1 targets. Of the ten security-related discoveries, four were classified as security vulnerabilities and six were categorized as general weaknesses with lower exploitation potential. It can be argued that this audit of the Psiphon Tunnel Core codebase revealed only a moderate number of flaws negatively affecting the scope. As such, the results testify to a solid security posture of the complex.

Supporting this verdict, no vulnerabilities of *Critical* nature were identified during *PSI-08*. Only one issue received a *High* score in terms of impact. Quite clearly, this issue should not be ignored and Cure53 advises treating it as a top priority. Specifically, the problem presents a way to identify the Psiphon servers (see PSI-08-008). Nevertheless, the overall impressions are positive, as the targets clearly received security attention from a skilled and committed team at Psiphon.

The following sections first describe the scope and key test parameters, as well as how the WP was structured and organized. Next, all findings are discussed in grouped vulnerability and miscellaneous categories. Flaws assigned to each group are then discussed chronologically. In addition to technical descriptions, PoC and mitigation advice will be provided where applicable.

The report closes with drawing broader conclusions relevant to this spring 2024 project. Based on the test team's observations and collected evidence, Cure53 elaborates on the general impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the Psiphon Tunnel Core codebase.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pentests & audits against Psiphon Tunnel Core codebase**
  - **WP1:** White-box pentests & audits against Psiphon Tunnel Core codebase
    - **Source code repository:**
      - **URL:**
        - https://github.com/Psiphon-Labs/psiphon-tunnel-core
      - **Commit:**
        - a77d3ca70bafcf6bcf060ea7337a7dee01fccbdc
    - **Out-of-scope aspects of the repository:**
      - **All** parts of the *inproxy* branch
        - https://github.com/Psiphon-Labs/psiphon-tunnel-core/compare/master...inproxy
  - **Client binaries were provided to Cure53 upon request**
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, each ticket has been given a unique identifier (e.g., *PSI-08-001*) to facilitate any future follow-up correspondence.

PSI-08-003 WP1: Fetching local URLs crashes *psiphond service* *(Medium)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

Under certain circumstances, the *psiphond* server crashes when resolving local URLs through the proxy. This is because the application is built statically with *netdns=cgo.* As such, the implementation can trigger an issue in *glibc*.

Even though these types of problems are known, they have not been addressed, neither by the *go* nor the *glibc* maintainers[1].

At the same time, the flaw could only be reliably reproduced when running a *psiphond* binary that was built on another system with a different version of *glibc*. These circumstances render real world occurrences less likely. Similarly, it remained infeasible to reproduce the issue with the officially released binaries.

Steps to reproduce:

1. Build *psiphond* from the GitHub repository. A static binary should be created.
2. Copy the binary to a different Linux system and run it.
3. On the client's side, access a local URL through the proxy:
   curl --proxy 127.0.0.1:8080 https://ads.local
4. The *psiphond* server crashes due to an issue in *glibc's getaddrinfo()* function.

Backtrace:

```
% ./psiphond run
[...]
SIGFPE: floating-point exception
PC=0x72163497250f m=7 sigcode=1
signal arrived during cgo execution
instruction bytes: 0x49 0xf7 0xf0 0x48 0x29 0xd1 0x48 0x89 0xf2 0x48 0x8d
0x84 0xe 0x0 0x8 0x0

goroutine 91 gp=0xc000103340 m=7 mp=0xc00030a008 [syscall]:
runtime.cgocall(0x9c2340, 0xc00006cda8)
```

[1] https://github.com/golang/go/issues/30310

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
      /usr/lib/go/src/runtime/cgocall.go:157 +0x4b fp=0xc00006cd80
sp=0xc00006cd48 pc=0x406a0b
net._C2func_getaddrinfo(0xc00011f650, 0x0, 0xc00057e660, 0xc00041c0f8)
      _cgo_gotypes.go:105 +0x55 fp=0xc00006cda8 sp=0xc00006cd80 pc=0x54ae55
net._C_getaddrinfo.func1(0xc00011f650, 0x0, 0xc00057e660, 0xc00041c0f8)
      /usr/lib/go/src/net/cgo_unix_cgo.go:78 +0x7a fp=0xc00006cdf0
sp=0xc00006cda8 pc=0x54b21a
[...]
```

It should be investigated why this problem is present, especially as it happens when the sources are compiled from scratch but not in the official binaries. What is more, Psiphon should consider using the build option *netdns=go* rather than *netdns=cgo.* This would help mitigate this issue.

PSI-08-007 WP1: Partial DoS of *meek* connections via buffer monopolization *(Low)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

The Psiphon solution complex offers several protocols to evade censorship and establish secure connections. These protocols include a variant of the pluggable transport *meek*[2], also available in the Tor browser.

In case a Psiphon server uses *meek* as its transport protocol, *meek* connections utilize two kinds of buffers for keeping responses in memory. One of the buffers is exclusive to the connection and has the size of 65kB. For responses larger than 65kB, all connections share a buffer pool which amounts to roughly 135MB. The purpose of keeping responses in memory is to cache them in case of connection failures noticed for clients.

Sharing the buffer pool has consequences, since an attacker who is a legitimate user of the Psiphon server could mount a partial Denial-of-Service attack against other users via *meek* as a transport protocol. For that purpose, the attacker monopolizes all memory from the shared buffer pool by producing traffic resulting in very large responses. This binds all memory offered through the buffer pool to the attacker, resulting in a DoS for responses larger than 65kB for legitimate users. In other words, the affected users would not be able to write them to the shared buffer pool from then on.

The code excerpt below demonstrates that the *meek* server uses the shared pool of memory from the field *server.bufferPool* on creating a new *meek* session of a client.

Affected file:
psiphon-tunnel-core-master/psiphon/server/meek.go

[2] https://blog.torproject.org/how-use-meek-pluggable-transport/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
func (server *MeekServer) getSessionOrEndpoint(
	request *http.Request, meekCookie *http.Cookie) (string,
		*meekSession, net.Conn, string, *GeoIPData, error) {
	[...]
	bufferLength := MEEK_DEFAULT_RESPONSE_BUFFER_LENGTH
	if server.support.Config.MeekCachedResponseBufferSize != 0 {
		bufferLength =
		server.support.Config.MeekCachedResponseBufferSize
	}
	cachedResponse := NewCachedResponse(bufferLength, server.bufferPool)

	session = &meekSession{
		meekProtocolVersion: clientSessionData.MeekProtocolVersion,
		sessionIDSent:       false,
		cachedResponse:      cachedResponse,
		cookieName:          meekCookie.Name,
		contentType:         request.Header.Get("Content-Type"),
	}
	[...]
	server.sessionsLock.Lock()
	server.sessions[sessionID] = session
	server.sessionsLock.Unlock()
	[...]
}
```

To mitigate this issue Cure53 advises creating individual buffers for individual connections. These items should not be shared across connections.

PSI-08-008 WP1: Replaying client preamble identifies Psiphon servers *(High)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

In order to make sure that Psiphon servers cannot be easily identified or enumerated, replay attacks against obfuscated protocols have been accounted for. Specifically, their mitigation entails using a random *nonce* for each connection.

This deployment is important as the strategy to prevent attackers from identifying Psiphon servers via response and traffic patterns. Unfortunately, the measure has not been implemented sufficiently, making it possible to replay obfuscated packets within a limited period. As the server responds to these replayed packets with mostly static data streams where only 30 bytes vary on each attempt, it gives enough reason to believe that is running a Psiphon service.

For possibly malicious actors, including governments, this is particularly interesting when individuals are under suspicion, but further evidence is needed for a more thorough observation. The steps below show how this issue can be reproduced. However, it should

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

be noted that the replay attack must be conducted from the IP address of the original client, as the code would successfully catch the reuse attack otherwise. This makes the issue a little bit harder to exploit for attackers with limited capabilities.

Steps to reproduce:

1. Set up a server and client with *OSSH* on port 9999.
2. With the server stopped, listen on port 9999 using *netcat* and write received data to a file at *nc -lvp 9999 > test*
3. Start the client which connects to port 9999 and stop it after *netcat* receives the connection.
4. Start the Psiphon server.
5. Send the previously recorded data to *psiphond* listening on port 9999, specifically using *cat test | nc localhost 9999*
6. Repeat *Step 5* a few times and observe the results.

The code below gets invoked when the client sends the preamble of an obfuscated stream and checks the received seed against a list of recently used values.

Affected file:
psiphon-tunnel-core/psiphon/common/obfuscator/obfuscator.go

Affected code:
```
if config.SeedHistory != nil {
	// Adds the seed to the seed history only if the magic value is
valid.
	// This is to prevent malicious clients from filling up the history
cache.
	ok, duplicateLogFields := config.SeedHistory.AddNew(
		config.StrictHistoryMode, clientIP, "obfuscator-seed",
osshSeed)
	errStr := "duplicate obfuscation seed"
	if duplicateLogFields != nil {
		if config.IrregularLogger != nil {
			config.IrregularLogger(
				clientIP,
				errors.BackTraceNew(errBackTrace, errStr),
				*duplicateLogFields)
		}
	}
	if !ok {
		return nil, nil, nil, nil, errors.TraceNew(errStr)
	}
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The highlighted section of the code below is reached when a new connection with a known seed value is established. If the *strictMode* parameter was set to *true,* the returned code would actually prevent this issue. Since this option is not set in the codebase, it defaults to *false*.

Affected file:
psiphon-tunnel-core/psiphon/common/obfuscator/history.go

Affected code:

```
func (h *SeedHistory) AddNew(
	strictMode bool,
	clientIP string,
	seedType string,
	seed []byte) (bool, *common.LogFields) {

	return h.AddNewWithTTL(
		strictMode, clientIP, seedType, seed,
lrucache.DefaultExpiration)
}
[...]
func (h *SeedHistory) AddNewWithTTL(
	strictMode bool,
	clientIP string,
	seedType string,
	seed []byte,
	TTL time.Duration) (bool, *common.LogFields) {

[...]

	previousClientIP, ok := h.seedToClientIP.Get(key)
	if ok {
		if clientIP == previousClientIP.(string) {
			logFields["duplicate_client_ip"] = "equal"
			return !strictMode, &logFields
		} else {
			logFields["duplicate_client_ip"] = "unequal"
			return false, &logFields
		}
	}

	logFields["duplicate_client_ip"] = "unknown"
	return false, &logFields
}
```

It should be investigated why the *StrictHistoryMode* option is disabled, given that it effectively makes this attack possible. As the code actually accounts for replay attacks, it is recommended to turn this setting on by default.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

PSI-08-010 WP1: Potential fingerprinting of Psiphon servers via gQUIC *(Medium)*

***Note**: No changes have been made to address this issue, as the problem is already documented and Psiphon is working on phasing out QUIC support for production networks.*

Psiphon servers support the QUIC protocol for tunnel transport. To that end, the administrator generates configurations for both server and client, indicating that a tunnel uses QUIC. Psiphon servers support, in principle, two variants of QUIC, namely IETF and Google QUIC (gQUIC).

It was discovered that the QUIC protocol handler of a Psiphon server requires an obfuscation key, known by both the client and server, as a precondition to completing the initial handshake between them. The initial handshake involves obfuscation in terms of encryption and serves to authenticate legitimate clients before establishing QUIC connections. For IETF QUIC connections, the server does not provide a response in case a client sends a *client-hello* message that is obfuscated with the wrong obfuscation key. On the contrary, the QUIC server responds for gQUIC channels in case a client provides a *client-hello* message protected by the provenly wrong obfuscation key.

An attacker could use this observation to probe for a potential Psiphon server which uses QUIC in the following way. First, the attacker attempts to establish a connection using IETF QUIC with a random obfuscation key. In case the server fails to provide any response but rather drops the connection silently, the attacker sends another connection attempt using Google QUIC with another random obfuscation key. In case the server responds to this second connection attempt, the attacker successfully identifies a potential Psiphon server.

It must be noted that this issue was discussed with the Psiphon team. Over the course of this exchange, it became clear that this appears to be an indicator of a Psiphon server. At that junction, it was mutually agreed to cease further investigation of the behavior of other and unrelated QUIC server implementations, as well as their responses to Psiphon clients using the QUIC implementation of Psiphon for further differentiation. The unit-test below demonstrates the issue.

Unit-test:

```
func TestQUIC(t *testing.T) {
	serverIdleTimeout = 1 * time.Second
	irregularTunnelLogger := func(_ string, err error, _
common.LogFields) {}

	realObfuscationKey := prng.HexString(32)
	listener, err := Listen(
		nil,
		irregularTunnelLogger,
		"127.0.0.1:0",
		realObfuscationKey,
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                true)
        defer listener.Close()

        if err != nil {
                t.Fatalf("Listen failed: %s", err)
        }
        serverAddress := listener.Addr().String()

        t.Run(fmt.Sprintf("OBFUSCATED-QUICv1(real-key)"), func(t *testing.T)
{
                runQUIC(t, "OBFUSCATED-QUICv1", false, serverAddress,
realObfuscationKey)
        })
        t.Run(fmt.Sprintf("OBFUSCATED-QUICv1(wrong-key)"), func(t *testing.T)
{
                runQUIC(t, "OBFUSCATED-QUICv1", true, serverAddress,
prng.HexString(32))
        })
        t.Run(fmt.Sprintf("gQUICv44(wrong-key)"), func(t *testing.T) {
                runQUIC(t, "gQUICv44", true, serverAddress,
prng.HexString(32))
        })
}

func runQUIC(
        t *testing.T,
        quicClientVersion string,
        shouldNotGetProbeResponse bool,
        serverAddress string,
        clientObfuscationKey string) {

        testGroup, testCtx := errgroup.WithContext(context.Background())
        testGroup.Go(func() error {

                ctx, cancelFunc := context.WithTimeout(
                        context.Background(), 1*time.Second)
                defer cancelFunc()

                remoteAddr, err := net.ResolveUDPAddr("udp", serverAddress)
                if err != nil {
                        return errors.Trace(err)
                }
                packetConn, err := net.ListenPacket("udp4", "127.0.0.1:0")
                if err != nil {
                        return errors.Trace(err)
                }
                packetConn = &countReadsConn{PacketConn: packetConn}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        obfuscationPaddingSeed, err := prng.NewSeed()
        if err != nil {
                return errors.Trace(err)
        }

        var clientHelloSeed *prng.Seed
        if isClientHelloRandomized(quicClientVersion) {
                clientHelloSeed, err = prng.NewSeed()
                if err != nil {
                        return errors.Trace(err)
                }
        }

        quicSNIAddress, _, err := net.SplitHostPort(serverAddress)
        if err != nil {
                return errors.Trace(err)
        }

        _, err = Dial(
                ctx,
                packetConn,
                remoteAddr,
                quicSNIAddress,
                quicClientVersion,
                clientHelloSeed,
                clientObfuscationKey,
                obfuscationPaddingSeed,
                nil,
                false)

        fmt.Println("Dial error: ", err)
        if err == nil {
                readCount := packetConn.
                        (*countReadsConn).getReadCount()
                fmt.Println("Read bytes from accepted con: ",
                        readCount)

                if shouldNotGetProbeResponse {
                        fmt.Println("Failed anti-probing for ",
                        quicClientVersion)
                        panic("Should not be raised")
                }
        }

        return nil
})

go func() {
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                testGroup.Wait()
        }()

        <-testCtx.Done()
        err := testGroup.Wait()
        if err != nil {
                t.Errorf("goroutine failed: %s", err)
        }
}
```

The unit-test establishes three connections to a Psiphon server using two different protocol variants, namely *OBFUSCATED-QUICv1* and *gQUICv44*. The goal of the tests is to verify the anti-probing measures of Psiphon servers. The first test for *OBFUSCATED-QUICv1* uses the correct obfuscation key, therefore the connection should be established successfully, with no *panic* raised.

The second test attempts to establish a connection using *OBFUSCATED-QUICv1* but now doing so with a truly random key. Here, the test should conclude that no connection was established, and no packets have been received by the test-client. The last test case attempts to establish a connection using *gQUICv44*, again with a random obfuscation key.

Test output:

```
=== RUN   TestQUIC
=== RUN   TestQUIC/OBFUSCATED-QUICv1(real-key)
Dial error:  <nil>
Read bytes from accepted con:  2
--- PASS: TestQUIC/OBFUSCATED-QUICv1(real-key) (0.00s)
=== RUN   TestQUIC/OBFUSCATED-QUICv1(wrong-key)
Dial error:  quic.Dial#494: quic.dialQUIC#1025: context deadline exceeded
--- PASS: TestQUIC/OBFUSCATED-QUICv1(wrong-key) (1.00s)
=== RUN   TestQUIC/gQUICv44(wrong-key)
Dial error:  <nil>
Read bytes from accepted con:  6
Failed anti-probing for  gQUICv44
panic: Should not be raised
```

From the output of the tests, it can be seen that the Psiphon server establishes a connection for *OBFUSCATED-QUICv1* with the correct obfuscation key, and fails to send a response for *OBFUSCATED-QUICv1* with a random key. The test for *gQUICv44,* however, demonstrates that the server sends six packets in response to the connection attempt, thereby failing to mitigate probing attacks.

To address this problem, Cure53 advises the provision of uniform responses for all QUIC protocol variants. This should become standard procedure in case a client provides a *client-hello* message protected with a wrong obfuscation key.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, whilst a vulnerability is present, an exploit may not always be possible.

PSI-08-001 WP1: No upper limit on sessions potentially leads to DoS *(Low)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

Clients connecting to a Psiphon server provide a session ID during the initial establishment of the SSH connection. After successfully verifying the client, the server creates a new session for that client with the provided session ID. Such a session includes several resources like TCP connection but also in-memory channels and threads for reading packets from the client. In this context, Cure53 confirmed that the server fails to impose a restriction on the maximum number of sessions it can manage.

An attacker could abuse this lack of resource restrictions. For example, they could attempt to establish numerous sessions on a single Psiphon server. When representing very powerful adversaries, like nation state-level threat actors, the attacker could potentially bring a Psiphon server into a Denial-of-Service (DoS) situation due to insufficient memory.

The excerpt below shows that the SSH server implementation, responsible for managing SSH sessions, fails to enforce an upper limit on the number of concurrent sessions.

Affected file #1:
psiphon-tunnel-core-master/psiphon/server/tunnelServer.go

Affected code #1:
```
func (sshServer *sshServer) registerEstablishedClient(client *sshClient)
bool {
       [...]
       if sshServer.clients[client.sessionID] != nil {
              [...]
       }
       [...]
       sshServer.clients[client.sessionID] = client
       [...]
}
```

Furthermore, it was identified that the handling of *meek* connections suffers from a similar issue. Specifically, as demonstrated in the code excerpt below, if the Psiphon server supports *meek* transport, the corresponding *meek* server (running on the Psiphon server) also fails to restrict the number of concurrent *meek* connections. As in the example above,

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

this could also potentially lead to a DoS situation similar to the one observed for the SSH connections.

Affected file #2:
psiphon-tunnel-core-master/psiphon/server/meek.go

Affected code #2:

```
func (server *MeekServer) getSessionOrEndpoint(
	request *http.Request, meekCookie *http.Cookie) (string,
*meekSession, net.Conn, string, *GeoIPData, error) {
	[...]
	session = &meekSession{
		meekProtocolVersion: clientSessionData.MeekProtocolVersion,
		sessionIDSent:       false,
		cachedResponse:      cachedResponse,
		cookieName:          meekCookie.Name,
		contentType:         request.Header.Get("Content-Type"),
	}
	[...]
	sessionID := meekCookie.Value
	if clientSessionData.MeekProtocolVersion >= MEEK_PROTOCOL_VERSION_2 {
		sessionID, err = makeMeekSessionID()
		if err != nil {
			return "", nil, nil, "", nil, errors.Trace(err)
		}
	}

	server.sessionsLock.Lock()
	server.sessions[sessionID] = session
	server.sessionsLock.Unlock()
	[...]
}
```

To mitigate this issue Cure53 recommends supplying a configurable limit of the number of concurrent sessions for a Psiphon server. Concrete limits should depend on the hardware settings that characterize individual Psiphon servers.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

PSI-08-002 WP1: Session impersonation through SSH API requests *(Medium)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

During the initial SSH handshake, a client sends their session ID together with the SSH password to a Psiphon server. After successfully authenticating the client, the server adds the client's session to its internal cache, and then starts threads for reading new messages from the client.

When reading SSH requests, which correspond to a special kind of API requests sent through a SSH connection, the server forwards these requests to its SSH API. This API includes several functions, for example those for completing a handshake. Cure53 observed that the server actually fails to cross-validate the session ID of the SSH API payload with the session ID of the connection that received the request.

In case an attacker manages to acquire a session ID of a victim, they could send SSH API requests through its own connection but using the session ID of the victim. This results in an impersonation of the victim for SSH API requests. More broadly, it could lead to other issues like session hijacking, DoS or information leakages.

The code excerpt below shows that the function responsible for handling SSH requests of a client fails to use the session ID of the *sshClient* instance. Furthermore, it is evident that the function does not check if the provided client session ID within the *request.Payload* instance matches the session ID of the *sshClient*.

Affected file:
psiphon-tunnel-core-master/psiphon/server/tunnelServer.go

Affected code:

```
func (sshClient *sshClient) handleSSHRequests(requests <-chan *ssh.Request)
{

	for request := range requests {
		[...]
		if request.Type == "keepalive@openssh.com" {
			[...]
		} else {
			[...]
			responsePayload, err = sshAPIRequestHandler(
				sshClient.sshServer.support,
				clientAddr,
				sshClient.geoIPData,
				authorizedAccessTypes,
				request.Type,
				request.Payload)
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                }
                [...]
        }

}
```

To mitigate this issue Cure53 advises implementing an additional cross-check. This mechanism should verify that the session ID of the SSH API request payload matches the session ID of the SSH client connection through which the server received the SSH API request.

PSI-08-004 WP1: OOB packet reading in *psiphond* tunnel handlers *(High)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

The Psiphon software complex permits usage of tunnel interfaces for establishing connections. For that purpose, a client creates a tunnel interface locally and sends outbound traffic through there. The client application of Psiphon, for example the *ConsoleClient*, picks up network data from this interface and sends it to the connected Psiphon server.

In response, the server sends the packet through its own tunnel interface. When an upstream service responds to the Psiphon server's tunnel interface, the server reads the packet from the tunnel interface and sends it downstream to the client. In this context, it was found that the packet parsing of the tunnel interface fails to validate the length of raw TCP packets. Moreover, it makes implicit assumptions about the lengths of the packets.

Failing to validate the lengths of packets can result in numerous complications. For buffers shared across connections with sufficient lengths, it could lead to out-of-bounds reads across client connections, potentially leaking connection information across clients. Another case would concern the application which allocates or slices buffers according to the observed yet too short lengths. If it additionally assumes a certain but unmet minimum length, the application will read out-of-bounds. This would signify a *panic* that results in a shutdown of the server.

It must be noted that packets and their length are out of control for a Psiphon server, and are potentially fully and maliciously controlled by upstream services. The excerpt below demonstrates the issue for IPv4 TCP packet parsing. It is evident that - for TCP connections - the minimum packet length enforced by the server corresponds to 34 bytes. However, the server's interface attempts to access the packet array at indices *[36:38]* (TCP), which are clearly out-of-bounds for packets that have the length of 34.

The team attempted to verify this dynamically by debugging the *psiphond* server. Instead of the expected *panic*, the testers encountered out-of-bounds reads without *panic*, which meant reading data from previous connections.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file:
psiphon-tunnel-core-master/psiphon/common/tun/tun.go

Affected code:

```
func processPacket(
	metrics *packetMetrics,
	session *session,
	clientTransparentDNS *clientTransparentDNS,
	direction packetDirection,
	packet []byte) bool {
	[...]
	if version == 4 {
		[...]
		if len(packet) < 20 {
			metrics.rejectedPacket(direction, packetRejectLength)
			return false
		}
		[...]
		protocol = internetProtocol(packet[9])
		[...]
		if protocol == internetProtocolTCP {
			if len(packet) < 33 {
				metrics.rejectedPacket(direction,
				packetRejectTCPProtocolLength)
				return false
			}
			dataOffset = 20 + 4*int(packet[32]>>4)
			if len(packet) < dataOffset {
				metrics.rejectedPacket(direction,
				packetRejectTCPProtocolLength)
				return false
			}
		} else if protocol == internetProtocolUDP {
			dataOffset = 28
			if len(packet) < dataOffset {
				metrics.rejectedPacket(direction,
				packetRejectUDPProtocolLength)
				return false
			}
		} else {
			[...]
		}

		applicationData = packet[dataOffset:]
		[...]
		sourceIPAddress = packet[12:16]
		destinationIPAddress = packet[16:20]
		IPChecksum = packet[10:12]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                [...]
                if protocol == internetProtocolTCP {
                        TCPChecksum = packet[36:38]
                } else { // UDP
                        UDPChecksum = packet[26:28]
                }

        } else { // IPv6
                [...]
        }
        [...]
}
```

It must be noted that the same issue is also present for the IPv6 packet parsing functionality.

To mitigate this issue, Cure53 advises to rigorously verify that the lengths of packets align with the elements the servers access. Better controls are needed to prevent out-of-bounds access.

PSI-08-005 WP1: Checks of SSH public keys omit host validation *(Info)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

A source code review of the Psiphon client application revealed that the client - at its core - establishes an SSH connection to a Psiphon server instance. As part of establishing the SSH connection, the client verifies the public key of the server against a known, expected public key. Yet, the client application fails to also validate the IP address of the server, with its actions limited to checking the public key.

This circumstance poses a security issue in case an attacker manages to steal a private key of a Psiphon server. To succeed, they would also need to successfully trick the victim into connecting to a different IP address than the expected Psiphon server address. In such a case, the victim's client application would successfully authenticate against the bogus SSH server of the attacker. Notably, this is possible because the attacker utilizes the stolen private key.

The code excerpt below demonstrates that the client application only compares the *publicKey* of a server rather than verifying the public key and the *remote* address of the server.

Affected file:
psiphon-tunnel-core-master/psiphon/tunnel.go

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
func dialTunnel(
	ctx context.Context,
	config *Config,
	dialParams *DialParameters) (_ *dialResult, retErr error) {
	[...]
	expectedPublicKey, err := base64.StdEncoding.DecodeString(
		dialParams.ServerEntry.SshHostKey)
	if err != nil {
		return nil, errors.Trace(err)
	}
	sshCertChecker := &ssh.CertChecker{
		IsHostAuthority: func(auth ssh.PublicKey, address string) bool
{
			[...]
			return false
		},
		HostKeyFallback: func(addr string, remote net.Addr, publicKey
ssh.PublicKey) error {
			if !bytes.Equal(expectedPublicKey, publicKey.Marshal())
{
				return errors.TraceNew("unexpected host public
key")
			}
			return nil
		},
	}
	[...]
}
```

Cure53 recommends to also validate the remote address as part of the SSH server key verification.

PSI-08-006 WP1: Weak MAC ciphers in SSH configuration *(Low)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

While dynamically analyzing the SSH configuration of Psiphon servers and its clients, the testers identified support for multiple authentication schemes in terms of message authentication codes (MACs). Among others, the configuration supports the *hmac-sha1-96* algorithm, which is considered insecure due to its rather short length in relation to the HMAC authentication tags[3].

Using a message authentication code with truncated length introduces a security risk of broken message authentication, as it increases the probability of an attacker successfully breaking the MAC. Subsequently, breaking the authentication scheme makes it possible for

[3] https://www.virtuesecurity.com/kb/ssh-weak-mac-algorithms-enabled/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

an attacker to alter encrypted messages without any of the parties - neither client nor server - ever noticing.

The code excerpt below demonstrates that the supported MAC modes of a Psiphon server include the insecure *hmac-sha1-96* algorithm.

Affected file:
psiphon-tunnel-core-master/psiphon/common/crypto/ssh/mac.go

Affected code:

```
var macModes = map[string]*macMode{
	"hmac-sha2-512-etm@openssh.com": {64, true, func(key []byte)
hash.Hash {
		return hmac.New(sha512.New, key)
	}},
	"hmac-sha2-256-etm@openssh.com": {32, true, func(key []byte)
hash.Hash {
		return hmac.New(sha256.New, key)
	}},
	"hmac-sha2-512": {64, false, func(key []byte) hash.Hash {
		return hmac.New(sha512.New, key)
	}},
	"hmac-sha2-256": {32, false, func(key []byte) hash.Hash {
		return hmac.New(sha256.New, key)
	}},
	"hmac-sha1": {20, false, func(key []byte) hash.Hash {
		return hmac.New(sha1.New, key)
	}},
	"hmac-sha1-96": {20, false, func(key []byte) hash.Hash {
		return truncatingMAC{12, hmac.New(sha1.New, key)}
	}},
}
```

To mitigate this issue Cure53 advises removal of the *hmac-sha1-96* from the list of the supported MAC algorithms. Only strong cryptographic MAC modes with full authentication tag lengths should be used as MAC schemes.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

PSI-08-009 WP1: Missing integrity check for obfuscated traffic *(Low)*

***Note**: The issue was fixed and the fix was verified by Cure53 who had access to the diff.*

Investigating the obfuscated SSH transport scheme revealed that the obfuscation applied during the initial handshake used the RC4 cipher to hide information from a potential eavesdropper. The encryption key for the cipher uses a pre-shared master secret for derivation. In this realm, obfuscation strategies fail to protect the integrity of the obfuscated data, as they solely focus on protecting its confidentiality.

The format of the messages during the initial obfuscated SSH handshake are rather fixed. Due to the missing integrity protection, an attacker could alter bytes in the handshake packets without the victim noticing. For example, the attacker could flip bytes of the *specName* field which is exchanged during the original and obfuscated SSH handshake.

To reiterate, the code excerpt below highlights the use of RC4 as an obfuscation scheme. As the RC4 solely protects the confidentiality of data, but not its integrity, an attacker can alter the obfuscated traffic without either client or server being able to detect the malicious modification.

Affected file:
psiphon-tunnel-core-master/psiphon/common/obfuscator/obfuscator.go

Affected code:

```
func initObfuscatorCiphers(
	config *ObfuscatorConfig, obfuscatorSeed []byte) (*rc4.Cipher,
*rc4.Cipher, error) {

	clientToServerKey, err := deriveKey(obfuscatorSeed,
[]byte(config.Keyword), []byte(OBFUSCATE_CLIENT_TO_SERVER_IV))
	if err != nil {
		return nil, nil, errors.Trace(err)
	}

	serverToClientKey, err := deriveKey(obfuscatorSeed,
[]byte(config.Keyword), []byte(OBFUSCATE_SERVER_TO_CLIENT_IV))
	if err != nil {
		return nil, nil, errors.Trace(err)
	}

	clientToServerCipher, err := rc4.NewCipher(clientToServerKey)
	if err != nil {
		return nil, nil, errors.Trace(err)
	}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        serverToClientCipher, err := rc4.NewCipher(serverToClientKey)
        if err != nil {
                return nil, nil, errors.Trace(err)
        }

        return clientToServerCipher, serverToClientCipher, nil
}
```

Further, reviewing the QUIC obfuscation revealed that also the QUIC obfuscation applies an encryption scheme without any integrity protection. For QUIC, client and server utilize the ChaCha20 stream cipher for obfuscation, without the often used Poly1305 algorithm to protect the integrity of the data in question.

The code excerpt below highlights the use of ChaCha20 without Poly1305. Once again, the lack of integrity protection fosters an attack in between Psiphon client and server, resulting in modification of obfuscated data with no alerts.

Affected file:
psiphon-tunnel-core-master/psiphon/common/quic/obfuscator.go

Affected code:

```
func (conn *ObfuscatedPacketConn) readPacket(
        p, oob []byte) (int, int, int, *net.UDPAddr, bool, error) {
        [...]
        isObfuscated := true
        [...]
        if n > 0 {
                [...]
                if isObfuscated {
                        [...]
                        cipher, err :=
                        chacha20.NewCipher(conn.obfuscationKey[:],
                        p[0:NONCE_SIZE])
                        if err != nil {
                                return n, oobn, flags, addr, true,
                        errors.Trace(err)
                        }
                        cipher.XORKeyStream(p[NONCE_SIZE:], p[NONCE_SIZE:])
                        [...]
                }
        }
        [...]
}
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue Cure53 advises to either compute an HMAC checksum in addition to the encryption of data in transit, or to switch an authenticated encryption mode (e.g., Galois Counter Mode) characterized by using a *de facto* strong block cipher like AES. For ChaCha20 encryption, it is advisable to also use Poly1305 as an authentication scheme that is capable of protecting the obfuscated data's integrity.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As the short list of findings and the absence of *Critical* flaws suggest, Cure53 is happy to report that the Psiphon Tunnel Core codebase appears to be in good shape from a security perspective. Four testers involved in this spring 2024 assessment were impressed with the coding style and source code quality, which both clearly demonstrate that security work focused on improving protections has been done successfully prior to this *PSI-08* audit. Despite spotting ten flaws, Cure53 still commends the Psiphon team for their commitment to security.

To clarify, Cure53 has a long-standing cooperation with Psiphon. As such, this white-box April-May 2024 audit was informed by past projects, including *PSI-01* as the examination which tackled the Tunnel Core as well. From a meta-level perspective, it is crucial to underscore that the Psiphon software complex constitutes a client-server complex with the aim to obfuscate traffic for evading censorship. It supports a large variety of different protocols and tactics, many of which were subject to Cure53's scrutiny in earlier iterations (see *PSI-01* to *PSI-07* reports*)*.

At its heart, the protocols utilize SSH connections, which are potentially obfuscated, to establish a secure tunnel between client and server. The client and server wrap the SSH connection, depending on the configuration, in other protocols such as *meek* or QUIC. In scope of this assessment, both client and server were considered, with an explicit focus on the core components used by both applications.

To assess the security of the Psiphon complex, the team conducted dynamic tests paired with static code analysis in terms of source code reviews. It was straightforward for the team to build client and server applications, however, the configuration of the system turned out to be rather complicated. This was generally due to the complexity of the product(s) and lack of documentation regarding some of the configuration options. While no major roadblocks were encountered, it is important to keep this in mind.

The Cure53 team started with conducting tests dedicated to SSRF issues that could affect the Psiphon servers. It was investigated if servers allow forwarding of requests into the internal network. A source code review highlighted the impossibility of this task, barring certain conditions. In particular, the intervention of an admin who allows such traffic through the *AllowBogon* flag could change the outcomes of SSRF attack attempts. At any rate, the deployment was considered correct for the use-case.

It is clear that Psiphon servers correspond to valuable targets for censors, therefore the identification of a potential Psiphon server imposes an immediate risk for a server. Hence, the testing team was investigating ways to find a Psiphon server through traffic patterns, referred to as fingerprinting.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

It turned out that the QUIC protocol implementation is vulnerable to fingerprinting when alternating between supported QUIC protocol variants, for instance when moving between "(obfuscated) IETF QUIC" to "Google QUIC". The server responds differently in case that the attacker does not know the obfuscation key of the connection, fostering adversarial mapping of potential Psiphon servers (PSI-08-010).

A focused inspection of the obfuscation protocol further revealed that MitM attackers could use recorded sessions to fingerprint Psiphon servers by replaying initial packets and observing the responses. Even though better code choices could mitigate this problem, the corresponding option was disabled at the time of testing (see PSI-08-008).

The Psiphon applications use operating system commands to configure their respective interfaces for some settings. Cure53 wished to know if such configurations could lead to the execution of arbitrary commands, however, no issues in this direction could be spotted.

Since the ultimate purpose of Psiphon servers is to relay requests of clients, availability was considered a fundamental requirement for the setup. Therefore, the team comprehensively analyzed the source code for potential Denial-of-Service situations. The envisioned scenarios included unhandled *panic*s (for example, by accessing *nil* pointers or reading out-of-bounds) and unbounded memory allocations.

Indeed, it turned out that Psiphon servers suffer from several issues in this area. For instance, the issue of PSI-08-007 highlights a partial DoS for *meek* connections, whereas the issue of PSI-08-004 describes an out-of-bounds read issue for Psiphon servers using tunnel interfaces. Lastly, PSI-08-001 sheds light on the missing checks in regard to the number of concurrent sessions held by Psiphon servers.

Another Denial-of-Service issue can occur when local hostnames are resolved through the tunnel, as described in PSI-08-003. However, this issue relies on certain conditions and could not be reproduced using the official binaries.

As the SSH connections require authentication in terms of an SSH username and password, the team was investigating the authentication scheme that the applications rely on. Here, it must be positively noted no flaws transpired in the authentication scheme of SSH connections. Also, the team attempted to mount a Man-in-the-Middle attack on the SSH connection between client and server, however, the client successfully pins the server's public key.

Next, the team was checking for potential flaws in authorizing access to the Psiphon server. The authorization scheme - in terms of signed authorizations - was found to be robust against threats like signature bypasses, the use of expired authorizations or sharing them with other clients.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As a topic closely related to authorization, the team was investigating impersonation attacks against other legitimate users of a Psiphon server. It was identified that an attacker could impersonate other legitimate clients through their own SSH tunnel. Solely knowing the SSH session ID of a victim would suffice for this approach, as can be seen in PSI-08-002.

The cryptography used by the complex was also investigated. The auditors checked for insecure randomness in key generations, outdated cryptographic primitives, and other common flaws like reusing cryptographic keys with constant IVs or non-constant time comparisons of secrets. Generally speaking, the cryptography appears sturdy, with just minor observations and recommendations ensuing.

For example, it was found that the obfuscation as part of SSH and QUIC connections fails to apply integrity checks on the obfuscated data (PSI-08-009). Moreover, SSH connections in principle allow the use of *hmac-sha1-96* as MAC scheme (PSI-08-006), which should be reconsidered.

All in all, it was concluded that the Psiphon complex is in a good state from a security standpoint. From the data gathered during *PSI-08,* it is evident from the coding style and the source code quality are underpinned by solid design choices and multiple iterations of security testing. Therefore, Cure53 congratulates the Psiphon team for a product with an excellent security posture.

Cure53 would like to thank Adam Kruger, Rod Hynes and Joe Arshat from the Psiphon Inc. team for their excellent project coordination, support and assistance, both before and during this assignment.