ARTIFACT EVALUATED
USENIX
AVAILABLE

Opossum Attack: Application Layer Desynchronization using Opportunistic TLS

Robert Merget
Technology Innovation Institute

Nurullah Erinola
Ruhr University Bochum

Marcel Maehren
Ruhr University Bochum

Lukas Knittel
Ruhr University Bochum

Sven Hebrok
Paderborn University

Marcus Brinkmann
Ruhr University Bochum

Juraj Somorovsky
Paderborn University

Jörg Schwenk
Ruhr University Bochum

Abstract

Many protocols, like HTTP, FTP, POP3, and SMTP, were originally designed as synchronous plaintext protocols – commands and data are sent in the clear, and the client waits for the response to a pending request before sending the next one. Later, two main solutions were introduced to retrofit these protocols with TLS protection. (1) *Implicit TLS:* Designate a new, well-known TCP port for each protocol-over-TLS, and start with TLS immediately. (2) *Opportunistic TLS:* Keep the original well-known port and start with the plaintext protocol, then switch to TLS in response to a command like STARTTLS.

In this work, we present a novel weakness in the way TLS is integrated into popular application layer protocols through implicit and opportunistic TLS. This weakness breaks authentication, even in modern TLS implementations if both implicit TLS and opportunistic TLS are supported at the same time. This authentication flaw can then be utilized to influence the exchanged messages after the TLS handshake from a pure MitM position.In contrast to previous attacks on opportunistic TLS, this attack class does not rely on bugs in the implementations and only requires *one* of the peers to support opportunistic TLS.

We analyze popular application layer protocols that support opportunistic TLS regarding their vulnerability to the attack. To demonstrate the practical impact of the attack, we analyze exploitation techniques for HTTP (RFC 2817) in detail, and show four different exploit directions. To estimate the impact of the attack on deployed servers, we conducted a series of IPv4-wide scans over multiple protocols and ports to check for support of opportunistic TLS. We found that support for opportunistic TLS is still widespread for many application protocols, with over 3 million servers supporting both, implicit and opportunistic TLS at the same time. In the case of HTTP, we found 20,121 servers that support opportunistic HTTP across 35 ports, with 2,268 of these servers also supporting HTTPS and 539 using the same domain names for implicit HTTPS, presenting an exploitable scenario.

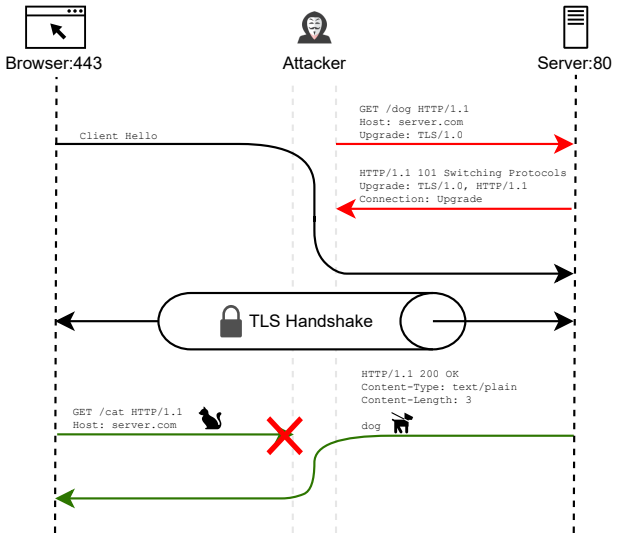

Figure 1: A sketch of the Opossum attack on HTTPS. Before the *ClientHello* of the browser reaches the server, the attacker establishes an HTTP connection and performs a `GET` request for `dog`, including a TLS upgrade header. The attacker then tunnels the browser's TLS messages through their HTTP connection. After the handshake, the client performs its `GET` request, which the attacker does not forward. At the same time, the server answers the request from the attacker, which the client then misinterprets as the answer to their request.

1 Introduction

Some of the earliest Internet protocols consist of simple, *synchronous*, text-based, plaintext communication. These protocols are still in widespread use today. Examples include HTTP up to version 1.1, FTP, POP3, SMTP, and LDAP. Since corresponding request-response pairs in these protocols are not labeled by a unique identifier, these protocols rely on responses arriving in the same order in which the requests were issued. Typically, these protocols are designed such that

a requesting client either waits for a response before sending the next request (e.g., HTTP 1.0 and SMTP) or sends a sequence of requests and expects the responses to arrive in the same order (e.g., HTTP 1.1 request pipelining and SMTP with command pipelining extension).

As a reliable network connection, TCP guarantees this synchronicity in each direction individually, even when random transmission errors occur. However, a Man-in-the-Middle (MitM) attacker can easily delete, modify, inject, or reorder responses by buffering data and adapting sequence numbers in the TCP headers. To prevent such attacks, SSL/TLS [21–23, 36, 39, 64, 65] was introduced. In general, there are two ways to integrate TLS into an existing application: *implicit* and *opportunistic* TLS.

Implicit TLS In an implicit TLS connection, the TLS channel is established *before* any application data is transmitted. This, again, can be done in two ways. Either a new, well-known TCP port can be reserved for the protocol-over-TLS variant (e.g., HTTPS/443 for HTTP/80). Or, for protocols where the client sends the first message, it is possible to use the protocol in its plain and encrypted variants on the same port: The server has to distinguish whether the first byte(s) sent by a client is plaintext or the start of a TLS *ClientHello*. This is feasible in HTTP, as employed by CUPS, because a plaintext HTTP request starts with an ASCII letter, while a *ClientHello* will begin with the binary byte `0x16` (record layer type set to "handshake") [23, 65].

Opportunistic TLS With opportunistic TLS, a TCP connection to the well-known TCP port of the application protocol is established first, and application data is sent. *After* this, establishing a secure TLS channel is requested within the application protocol. The application protocol is paused while the TLS handshake takes place over the established TCP connection, and it is resumed within the TLS record layer once the handshake is complete. Opportunistic TLS is backward compatible with deployed clients and firewalls as it reuses the same port as an insecure connection, and a connection can also succeed if the client or server does not support TLS.

The support for opportunistic TLS varies heavily based on the respective protocol and community using it. For SMTP, opportunistic TLS was introduced on a new port 587 [52], replacing the standard port 25, while implicit TLS was initially standardized on port 465 but later deprecated [40] in favor of opportunistic TLS. For IMAP, opportunistic TLS was added to the standard port 143 [40], and implicit TLS was standardized on the new port 993 [55]. For POP3, opportunistic TLS was added to the standard port 110 [40], and implicit TLS was standardized on the new port 995 [55]. For the opportunistic TLS variants of SMTP, IMAP, and POP3, TLS is activated using the STARTTLS command. For FTP, the only official method is opportunistic TLS, standardized in RFC 4217 [35], although implicit TLS is still supported in many applications, typically on port 990 [18]. Within these email and FTP protocols, opportunistic TLS is still widespread, with millions of servers supporting it [59].

In the context of HTTP(S) [63], implicit TLS is the norm (port 443). Opportunistic TLS for HTTP is specified in RFC 2817 [47], and is implemented in some web servers like Apache 2. Although it was never formally deprecated, it did not get adopted by browser vendors,[1] so its use on the World Wide Web is expected to be a rare occurrence. However, it was adopted by the Internet Printing Protocol (IPP, RFC 8010 [73]), XCAP (RFC 4825 [68]), and recommended for DHCPv6 network boot (RFC 5970 [41]). To the best of our knowledge, the usage and distribution of opportunistic TLS variants beyond FTP and email protocols have never been analyzed in an academic context.

Novel Authentication Weakness In this work, we demonstrate that supporting both opportunistic TLS and implicit TLS, even if both are supported only by the client or only by the server, exposes an authentication weakness. An attacker can misuse this weakness to influence the messages exchanged on the application layer from a pure MitM position, ultimately undermining the fundamental TLS security guarantee of integrity protection. The attacker can compromise the integrity of the secure channel across all TLS versions, regardless of all known countermeasures in place.

This weakness can then be used by an attacker to perform *desynchronization* in popular application-layer protocols. For HTTP, an attacker can abuse the flaw to inject a chosen (malicious) request, and the resulting (malicious) response will be delivered to the web browser *within the secure TLS channel*, which the browser interprets as the response to *its* request. All future request/response message pairs will remain unsynchronized. Given the nature of the weakness, only one communication peer needs to support both opportunistic and implicit TLS. In the context of HTTP, this means that even though *no single* web browser supports opportunistic TLS, the attack still works against HTTPS connections with HTTP servers that do.

Example for a Desynchronization Attack Against HTTPS We outline the concrete Opossum attack against HTTPS in Figure 1. Note that in HTTP, the client can ask for a connection to be upgraded to opportunistic TLS by adding an extra upgrade header to its first, unprotected, plaintext request (see Figure 2 in Section 2.1). To perform the attack on HTTP, the attacker waits for a victim client to initiate a server connection using implicit TLS. The attacker then opens a connection to the plaintext port of the server and establishes a TLS session using opportunistic TLS. The server now expects a *ClientHello* message, which the attacker forwards from the

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=276813, https://issues.chromium.org/issues/41154248

victim client to the server. The attacker forwards all further messages between the client and server, allowing them to perform a TLS handshake. After the handshake, both peers attempt to speak the application protocol over TLS, but in different variants: The client assumes the protocol uses implicit TLS and no messages were exchanged before the handshake. The server, however, assumes that opportunistic TLS is used and that messages were exchanged before the handshake. This creates a situation where both parties follow a slightly different application layer protocol. In the opportunistic variant, the server is the first to send a message after the TLS handshake (as a response to the upgrade request), while in the implicit variant, the client is the first to send a message. As a result, both parties attempt to send a message simultaneously, which the attacker can exploit by rearranging the response from the server to the attacker as a response to the client's request. The victim client misinterprets the response as belonging to its request.

In the example in Figure 1, the client interprets `dog` to be the response to the requested `/cat`. Although this example is specific to the case of HTTPS, where it has particularly severe consequences, we will demonstrate that, similarly, for many other protocols, the parties can also become desynchronized. After desynchronization, they disagree on which party sends the first message, as the application layer protocols of both variants do not exactly match. While we expect these vulnerabilities to be difficult to exploit in non-HTTP protocols, the attack breaks formal channel integrity assumptions in a similar way to the ALPACA attack [18].

HTTPS Desynchronization Attacks In the context of HTTPS, we show that desynchronization leads to four attack classes.

- **Resource Confusion:** The attacker tricks the web server to return the content of a different resource to the browser's request. For example, the attacker can replace an image with another image on the same server.
- **Session Fixation:** The attacker tricks the client into receiving the session cookie of the attacker, logging the user into the attacker's account.
- **Reflected XSS:** The attacker leverages a server-side reflected self-XSS vulnerability to escalate it into a regular reflected XSS.
- **Software-Specific Issues:** Implementation issues in the TLS upgrade mechanism allow an attacker to leak request headers, such as cookies.

We evaluate the prevalence of the Opossum attack by performing IPv4-wide scans for HTTP, FTP, IMAP, POP3, and SMTP regarding opportunistic and implicit TLS support. In total, we identified over 3 million hosts that support opportunistic TLS alongside implicit TLS on vulnerable application-layer protocols, thereby enabling the attack scenario. While we do not show exploits for all affected protocols, the Opossum attack is yet another attack targeting opportunistic TLS, highlighting the risk this feature introduced. We, therefore, join previous conclusions by Poddebniak et al. [59] and recommend switching away from opportunistic TLS to implicit TLS where possible.

Contributions Our main contributions are as follows:

- We show a novel authentication weakness, found in many opportunistic TLS standards, that allows the attacker to influence the exchanged application data messages.
- We demonstrate that this weakness can be exploited to break synchronization assumptions of the application-layer protocols.
- We analyze the impact of the discovered vulnerability regarding HTTPS in detail.
- We evaluate the impact on 43 ports across five protocols and estimate the number of affected services.

2 Background

The Transport Layer Security (TLS) protocol, formerly known as SSL, is one of the most important cryptographic protocols on the Internet. TLS establishes a secure connection between a client and a server that provides confidentiality, integrity, and authentication for the transmitted data. The TLS protocol exists in many versions; the most recent one is TLS 1.3 [65]. To establish a TLS connection, both parties first perform a handshake, which starts with the client sending a *ClientHello* message announcing its TLS version and supported cryptographic parameters. The server selects from the client's offered choices and responds with a batch of messages, starting with the *ServerHello* that announces the chosen parameters. Along with the *ServerHello*, the server sends a public key for a Diffie-Hellman key exchange, a certificate declaring its identity, and a signature to be verified using the certificate to *prove* its identity. Once both parties shared their public key, they transition into the encrypted state. Finally, each party sends a *Finished* message containing a cryptographic checksum of their session transcript to ensure the integrity of the exchanged handshake messages. Subsequently, both parties may send application data.

Channel Security After a TLS handshake, the client and server have established two separate secure channels: one channel is used for client-to-server messages, and one channel is used for server-to-client messages. The channels use symmetric encryption and MAC authentication, preventing MitM attacks such as injection, deletion, modification, and reordering of messages. While Smyth and Porenti [72], and

Bhargavan et al. [17] have shown that truncation attacks are still possible, they can be detected at the end of a connection by the lack of a terminating alert message (*close_notify*).

Opportunistic TLS Many application layer protocols were upgraded in hindsight to support TLS by switching to encryption in the middle of the execution. This allowed users to deploy the secure version and the plaintext version on the same port. To perform the upgrade, peers can usually send a specific command to trigger the upgrade. Once upgrading is agreed upon, the client starts the TLS handshake on the existing TCP connection. This upgrading mechanism is called *opportunistic TLS*, or STARTTLS. Many popular protocols support opportunistic TLS, such as HTTP [33, 47], FTP [35, 60], MySQL [12], PostgreSQL [14], NBD [61], SMTP [40, 48, 52], LMTP [56], IMAP [20, 55], POP3 [54, 55], NNTP [29, 31, 53], LDAP [70], Managed Sieve [50] and XMPP [69].

Opportunistic TLS is (intentionally) vulnerable to MitM downgrade attacks, as an attacker may convince both sides that TLS is not supported to keep the connection as plaintext. While this was an acceptable trade-off in the early days of adoption, modern clients and sometimes servers may refuse to use the connection for anything security-sensitive before the connection has been upgraded to TLS. In this work, we assume that implementations use this strict interpretation and will not leak any secrets to an attacker if TLS is not supported.

2.1 HTTP

The Hypertext Transfer Protocol (HTTP) was designed as a plaintext protocol for transferring hypertext documents across the Internet. This fundamental design choice, while enabling human readability and easier debugging, introduces several security implications. HTTP messages, including headers and payload data, are transmitted as plaintext ASCII, making them susceptible to interception and manipulation by malicious actors positioned between client and server.

World-Wide Web The main application of HTTP is exchanging HTML documents in the World-Wide Web (WWW). Web features like JavaScript allows servers to send Turing-complete code to the browser, which then executes this code. To limit the abilities of a malicious server, web browsers use the Same-Origin-Policy (SOP). In this model, the scope of what can be accessed through JavaScript is bound by the protocol, port, and domain of the server.

To execute JavaScript in Web applications, attackers can leverage Cross-Site Scripting (XSS) attacks. In an XSS attack, the attacker injects malicious scripts directly into a trusted website. The attack exploits the browser's inability to distinguish between legitimate and malicious script sources. An attacker who can introduce malicious JavaScript into the context of a trusted website, can use JavaScript to potentially steal session tokens, cookies, or other sensitive data.

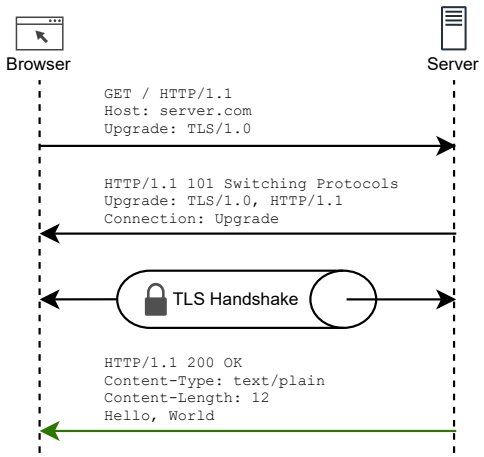

Figure 2: Message flow of an HTTP connection with TLS upgrade. The client proposes to upgrade the connection. The server can accept the upgrade request by sending an HTTP response code `101 Switching Protocols`. The client then initiates a TLS handshake. After the TLS handshake, the server sends the requested resource secured over the established TLS channel.

HTTPS HTTPS is an extension of HTTP that uses implicit TLS instead of plaintext HTTP to secure its connections. While nowadays, the overhead of this additional encryption is often negligible, in the mid-to-late 1990s, when it was first adopted, the additional computation time was a real consideration. The deployment of HTTPS required opening a dedicated port, and many websites served their content in mixed setups, where only some parts of the website were protected by TLS, while others were not. This caused many websites to support both HTTP and HTTPS on separate ports. Because support for HTTPS was not universal, users often initiated a connection to a website using HTTP and then switched to HTTPS by following a redirect from the server (e.g., as part of a login process).

Upgrading to TLS Within HTTP/1.1 In the early days of HTTPS adoption, RFC 2817 [47] was proposed, which uses *opportunistic* encryption to upgrade an HTTP connection to HTTPS, reusing the same port without establishing a new TCP connection. To do this, the *upgrade* header was introduced (see Figure 2). A client requesting a resource can include the upgrade header (`Upgrade: TLS/version`), requesting the server to optionally serve the response over TLS. To accept the request, the server answers with an HTTP status code `101 Switching Protocols` and an appropriate upgrade header. Once a client receives this status code with the upgrade header, the TLS handshake starts. After the handshake, the server sends the initially requested resource protected by TLS. Notably, the TLS version in the upgrade request is empirically ignored,

as TLS version negotiation is done through the handshake protocol instead.

The HTTP-to-TLS upgrade was only envisioned as opportunistic encryption, meaning it should only protect against passive attackers. An active attacker can remove the upgrade header from the client's request, causing the connection to stay unencrypted. Even if the server enforces that the connection is secured via TLS, a MitM attacker can act as a translator between the browser and the server, breaking the security guarantees.

The HTTP-to-TLS upgrade feature was never widely supported by web browsers but was adopted by the Internet Printing Protocol (RFC 8010 [73]), XCAP (RFC 4825 [68]), and recommended for DHCPv6 network boot (RFC 5970 [41]).

2.2 Protocol Negotiation

ALPN Extension The TLS protocol introduced an extension, the Application Layer Protocol Negotiation extension (ALPN) [37], to multiplex different protocols on the same port. With the ALPN extension, the client can send the application layer protocols it is willing to speak after the TLS handshake in its *ClientHello*, and the server can then respond with the selected application layer protocol in its *ServerHello* message.

ALPACA Attack The ALPACA attack [18] exploits a weakness in the authentication mechanism of TLS to confuse peers with which server endpoint they are talking to. Since TLS does not protect the port numbers or IP addresses, an attacker can forward the TLS messages from the client intended for IP and port to a server running on a different IP and port, as long as the certificate that the server provides is also valid for the other server. This can result in a scenario where the client and the server are speaking different application layer protocols, which can again result in the loss of sensitive information. The authors of the ALPACA attack proposed *strict ALPN verification* as a countermeasure, which was later adopted by RFC 9325 [71]. This means that if the server and the client disagree on an application layer protocol through ALPN, the server should terminate the connection.

3 Opossum Attack

In this work, we show a new weakness in the way the TLS protocol is integrated into many popular application layer protocols, by supporting both implicit and opportunistic TLS at the same time. This weakness formally breaks authentication and channel integrity guarantees of TLS. We will then show that this weakness is actually exploitable in a new attack called *Opossum attack*, to *desynchronize* the communication in many popular application layer protocols.

3.1 Model

Attacker Model The Opossum attack considers a MitM attacker scenario where the attacker can read and manipulate all transmitted data at the TCP level. We assume the attacker cannot break the cryptographic primitives used in the TLS connection. The attack does not rely on software bugs, and all peers behave standard-compliant. The attack does not rely on a specific TLS version and works on all current TLS versions, even when client authentication is enabled.

Authentication Weakness The core insight from the ALPACA attack was that TLS certificates do not contain enough information to uniquely identify a server. In the context of HTTP, identities on the Internet typically use the *web origin*. A web origin is the combination of <protocol, domain, port>. However, TLS does not protect the intended web origin, as it has no means to securely communicate the client's intent to the server. An application server, therefore, cannot distinguish between requests sent to <http, domain, 80> and requests sent to <https, domain, 443>, and will therefore perform the handshake with the client. For example, if two different application servers are hosted on the same domain name, an attacker can redirect traffic from one server to the other without either party noticing that an attack has occurred. The proposed countermeasure (which was adopted in RFC 9325 [71]) was to strictly verify ALPN and SNI strings and fail on the server side if a mismatch is detected. Especially strict ALPN verification was supposed to ensure that both parties actually speak the same application layer protocol, meaning that the most severe form of confusion would be prevented, as the semantics of the protocol are at least preserved.

However, the ALPACA attack and the proposed countermeasures did not account for subtle differences in the way opportunistic TLS was integrated into many application-layer protocols. Protocols using opportunistic TLS use the same ALPN identifiers as implicit TLS variants, but sometimes behave slightly differently after a TLS handshake, making it formally a different application-layer protocol.

This allows the Opossum attack to still perform a 'cross'-protocol attack, similarly to ALPACA, where both endpoints are using different variants of the same application protocol, which in turn influences the exchanged application data messages between the endpoints. This formal weakness is present in many protocols that support opportunistic TLS, and as we will demonstrate in Section 4, can be leveraged to construct fully functional exploits for HTTP.

3.2 Attack Scenarios

For the attack, we consider two scenarios where client and server will end up using different invocation mechanisms for TLS: The client will use implicit TLS and the server opportunistic TLS (I2O, Figure 3), and vice versa (O2I Figure 4).

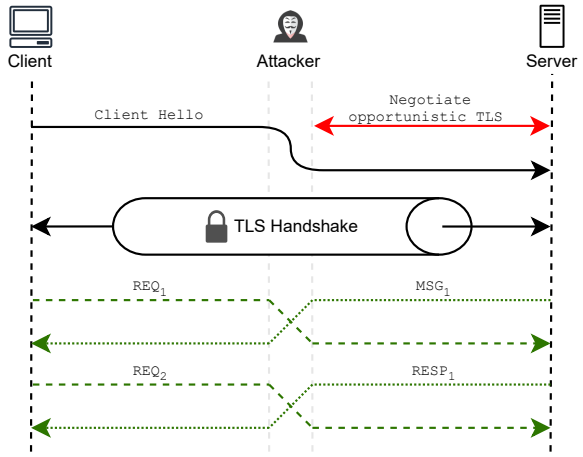

Figure 3: Desynchronization attack in the I2O scenario. Different line types denote unidirectional TLS channels.

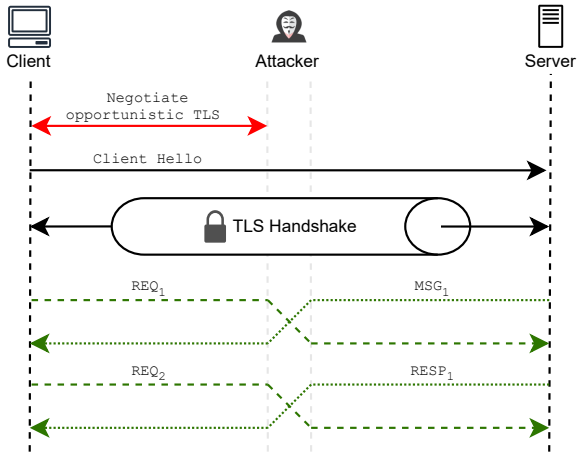

Figure 4: Desynchronization attack in the O2I scenario. Different line types denote unidirectional TLS channels.

To keep the attack scenarios focused on practically relevant scenarios, we only consider *synchronous* protocols. A synchronous protocol requires that responses arrive in the same order as the requests were sent. This description applies to many popular text-based protocols, such as HTTP 1.0/1.1, SMTP, POP3, and FTP.

Implicit-to-Opportunistic (I2O) In Figure 3, the client invokes a TLS handshake by sending the *ClientHello* message, with the intent to connect to the implicit TLS endpoint on the server. The MitM attacker delays this message and establishes a TCP connection to the opportunistic endpoint of the TLS server. The attacker then requests an upgrade to TLS (opportunistic). After the server has accepted this request, the attacker forwards the delayed *ClientHello* and all other handshake messages unaltered between the client and server. Since TLS establishes two separate unidirectional channels, the attacker may delay the forwarding of $RESP_1$ until they have received (and blocked) REQ_2 from the legitimate client. So the client receives an unexpected MSG_1 to its REQ_1. Since the client was expecting to receive the response to its request, the client interprets MSG_1 as $RESP_1$, and, if the connection continues, $RESP_1$ as $RESP_2$, and so on, causing a *desynchronization* of the connection. Of all the protocols analyzed in this paper, only HTTP falls into this category.

Opportunistic-to-Implicit (O2I) In the protocols that fit Figure 4, the client typically expects a greeting from the server after the TCP connection has been established. This may be some status message like "`220 example.org SMTP server ready`" in SMTP. While the connection is still plaintext, the attacker can act as the server and send banner messages or other required protocol-compliant messages to negotiate the usage of opportunistic TLS. Once negotiated, the client starts the TLS handshake, and just as in the I2O example, the attacker simply forwards the messages between the client and the (this time implicit) TLS server, translating between the ports. Once the connection is established, the client continues to use the application and sends a REQ_1. The server, which thinks that it is part of a regular implicit TLS connection, sends MSG_1, which is the welcome message of the implicit connection. Just like in the I2O example, this creates a situation where both parties are sending a message at the same time. This causes the client to treat MSG_1 as the response to REQ_1. This O2I attack scenario applies to protocols like FTP, LMTP, NNTP, POP3, and SMTP.

Please note that in both Figure 3 and Figure 4, the attacker can repeat the process of delaying responses from the server indefinitely until either peer stops responding.

3.3 Applicability

We investigated different synchronous protocols regarding their support for both implicit and opportunistic TLS (Table 1). Protocols like MySQL, NBD, and XMPP, which only support opportunistic TLS, were excluded from further study. For the remaining protocols, we studied the protocol flow in the specifications. For some protocols (IRC, LDAP, RDP, Managed Sieve, PostgreSQL) the Opossum attack does not apply, as the protocol flow after the TLS handshake is the same for both implicit and opportunistic TLS.

For the seven remaining protocols from Table 1, we identified discrepancies on the application layer after an implicit and opportunistic connection. We verified these discrepancies by providing a Proof-of-Concept (PoC) with a single client implementation and a single server implementation

We could verify that the PoCs for all seven protocols selected for verification worked, and that indeed, both parties tried to send the first message after the TLS handshake. For

one protocol, HTTP 1.1, we went further and built several exploits. Five more protocols are classified as *affected* in Table 1. We mark a protocol as affected when the protocol after the TLS workflow is different, as it allows an attacker to influence the exchanged message beyond what it should be able to. However, we did not investigate exploit strategies for these protocols and merely stated that a desynchronization is possible. IMAP is in a separate category since it is technically affected but not a synchronous protocol. Instead, requests and responses are *tagged* with a unique identifier, and responses can be returned in any order. It suffers from the weakness but cannot be desynchronized.

Our practical exploitation of HTTP 1.1 is described in detail in Section 4. These exploits follow the I2O attack (Figure 3). For HTTP servers, we investigate the support for opportunistic TLS; the results are given in Section 4.2. Other protocols are further studied in Section 5.

4 HTTPS Desynchronization with Upgrade Headers

The HTTP 1.1 protocol is a synchronous protocol, where the client sends the first message. When using implicit TLS, the client still sends the first message after the TLS handshake. However, when using the opportunistic variant, the server sends the first message after the TLS handshake. The attacker can trigger opportunistic TLS by sending an `Upgrade: TLS/1.0` HTTP header (cf. Figure 1). This corresponds to the I2O scenario presented in Figure 3 and does not require the client to support opportunistic TLS. In contrast to other protocols analyzed in this paper, HTTP uniquely allows the attacker to partially control the message the server sends after the TLS handshake by choosing a malicious plaintext request. In this section, we explore how an attacker can take advantage of a desynchronization in HTTP caused by the Opossum attack.

4.1 Man-in-the-Browser (MitB)

For the attacks on HTTP, we consider the same attacker model as for the general Opossum attack (Section 3). Additionally, in some cases, we also assume a Man-in-the-Browser (MitB) attacker model, where the attacker can additionally execute JavaScript code *outside* the targeted website's scope. This attacker model is frequently considered in the context of TLS attacks [26, 51, 67, 74].

4.2 Software Supporting Opportunistic HTTP

We *manually* searched the Internet for software that supports opportunistic HTTP and is thus susceptible to our attack. Our process included examining GitHub repositories and project sites, reviewing official documentation, and analyzing server headers from our HTTP study (Section 6). In the following sections, we list and describe the various tools and libraries we identified that implement opportunistic TLS.

Apache The Apache webserver [1] supports opportunistic HTTP since version 2.1 from 2005. To enable it, a user has to set the parameter `SSLEngine optional`.[2] Apache's implementation behaves non-standard conform when facing HTTP requests with an upgrade header with a non-zero content length. RFC 2817 expects the whole initial request to be sent in plaintext, while Apache expects only the header of the request to be sent in plaintext, with the body of the request being sent encrypted *before* the server sends the response. The problem with this behavior is that it assumes that the client will already know that the upgrade will be successful when it sends the HTTP header to trigger the upgrade, as the client must hold back the payload of the request to send it afterward. However, a server that does not support or accept the upgrade will not initiate the switching to another protocol but will instead wait for the body of the request, resulting in a deadlock. While on the surface, encrypting the POST data seems more secure, it actually results in a much more severe desynchronization, as we show in Section 4.4 and Figure 5.

Printer CUPS, a popular Unix application for printing services, supports the Internet Printing Protocol (IPP), for which CUPS always supports opportunistic and implicit TLS. Neither opportunistic nor implicit TLS can be disabled.[3] The CUPS server ignores any other HTTP headers from HTTP upgrade requests and, therefore, does not consider the content length that is provided.

PAPPL [13] is a simple C-based framework for developing CUPS printer applications. PAPPL supports HTTP-to-TLS upgrade as a client. Based on PAPPL or CUPS, a lot of printer applications also support opportunistic TLS, including LPrint [11], HP [7], HPLIP [8], Gutenprint [6], Ghostscript [5], and PostScript [15].

HttpClient 5 HttpClient is a component of the Apache HttpComponents project that provides a powerful HTTP client library. It supports an HTTP-to-TLS upgrade on the client side. In earlier versions, this upgrade feature was enabled by default. However, with version 5.4, the upgrade functionality has to be enabled explicitly.[4]

Icecast Icecast is an open-source streaming media server. The Icecast-Server [9] also supports[5] opportunistic HTTP on the server side, while the Libshout library [10] implements opportunistic HTTP on the client side.

[2] https://httpd.apache.org/docs/2.4/mod/mod_ssl.html
[3] https://www.cups.org/doc/encryption.html
[4] https://issues.apache.org/jira/browse/HTTPCLIENT-2344
[5] https://wiki.xiph.org/Icecast_Server/known_https_restrictions

Cyrus IMAP Cyrus [3] is an email, contacts, and calendar server that implements a limited HTTP server (called `httpd`[6]) to support CalDAV, CardDAV, and WebDAV. This HTTP server implements opportunistic HTTP upgrades when TLS is enabled.

4.3 Impact Analysis

The impact of the attack is comparable to the Renegotiation attack [62, 66] on TLS, which allows the attacker to choose an arbitrary prefix for the stream of the client. In contrast, in the Opossum attack, the attacker cannot choose an arbitrary prefix but has to choose a complete HTTP request as a prefix. The attacker also cannot send incomplete messages, which makes the attack strictly weaker. Nevertheless, this still allows for multiple exploit avenues.

4.3.1 Resource Confusion

The first attack goal is to confuse the user at the application layer, as described in Figure 1. This attack allows for changing the content delivered to the user. In Figure 1, the user retrieves a `dog` response instead of a `cat`. An attacker could replace images or executable file downloads with other images or files hosted on the same server. The real attack impact may vary. For example, the attack could also replace a JavaScript source file, potentially downgrading the version of a JS library. This can have further security implications, as the attacker can abuse this to prevent security-critical JavaScript libraries, such as DOMPurify [38], from functioning correctly.

PoC – Cat-Dog Example with Apache We implemented the basic resource confusion attack from Figure 1. When requesting `cat.html`, a victim is served `dog.html` instead.

4.3.2 Session Fixation

With Opossum, it is possible to perform a session fixation attack. By logging in with attacker-controlled credentials in the first request, the server will answer by setting a cookie to the attacker's session. This causes the client to receive and store the cookie. The client includes the cookies in subsequent connections, logging them into a session under the attacker's credentials.

PoC – Anti-CSRF Token Fixation in CUPS We implemented a PoC attack against CUPS that exploits the Opossum attack to force a specific anti-CSRF cookie into a client. This enables CSRF attacks against the admin interface in CUPS.

[6] https://www.cyrusimap.org/imap/reference/manpages/systemcommands/httpd.html

4.3.3 Reflected-XSS

The Opossum attack amplifies the attack surface of Reflected-XSS vulnerabilities. Usually, Reflected-XSS involves the user clicking on a specifically crafted link that triggers the vulnerability and includes the payload. As the attacker can control the request, they can trigger the vulnerability in the first request, causing the client to receive the response containing the JavaScript payload. The client will then execute the payload in the context of the attacked website. Further, a user-clickable link can only cause a `GET` request and the attacker cannot change the HTTP headers. With Opossum, the attacker can also exploit vulnerabilities that require a different method (e.g., `POST`) or specific HTTP headers, which may usually not be exploitable.

PoC – Exploiting Range Headers for XSS Servers that support the HTTP *Range* request header [32] (e.g., Apache) allow clients to request specific parts of a response. In the PoC, we use the range header in the TLS upgrade request to extract a commented-out script tag from a response, leading to XSS. Additionally, it is possible to request multiple ranges, which may allow an attacker to craft nearly arbitrary responses. However, when multiple ranges are requested, Apache sets the content-type to `multipart/byteranges`, a format that in all modern browsers will trigger a download and not render.

4.4 Software-Specific Issues

Since Apache expects the request body of the request that initiates the upgrade to be sent after the TLS handshake, an even more powerful desynchronization attack is possible (Figure 5), that is no longer bound by the message boundaries but instead desynchronizes requests completely, leading to a request smuggling attack[7]. To perform the attack, the attacker uses the Opossum attack but also specifies a content length in the original HTTP upgrade request. When performing the Opossum attack, from the server's perspective, the first request from the implicit client will be interpreted as the body of the attacker's request. More severely, the remaining data of the client's request will be interpreted by the server as the subsequent incoming HTTP request. In a MitB attacker model, the attacker can control the request body of the client and include the headers of yet another HTTP request to gain full control of the desynchronization. The attacker can then use this request to again contain the body of the *next* victim request as the body, this time with an attacker-controlled prefix. This, for example, allows the attacker to perform a POST request of a client request to any resource, which in many applications allows the attacker to steal the cookie.

We also tested CUPS, Icecast, and Cyrus IMAP for this behavior, but all of them handle upgrade requests with content length differently than Apache. CUPS ignores the content

[7] https://cwe.mitre.org/data/definitions/444.html

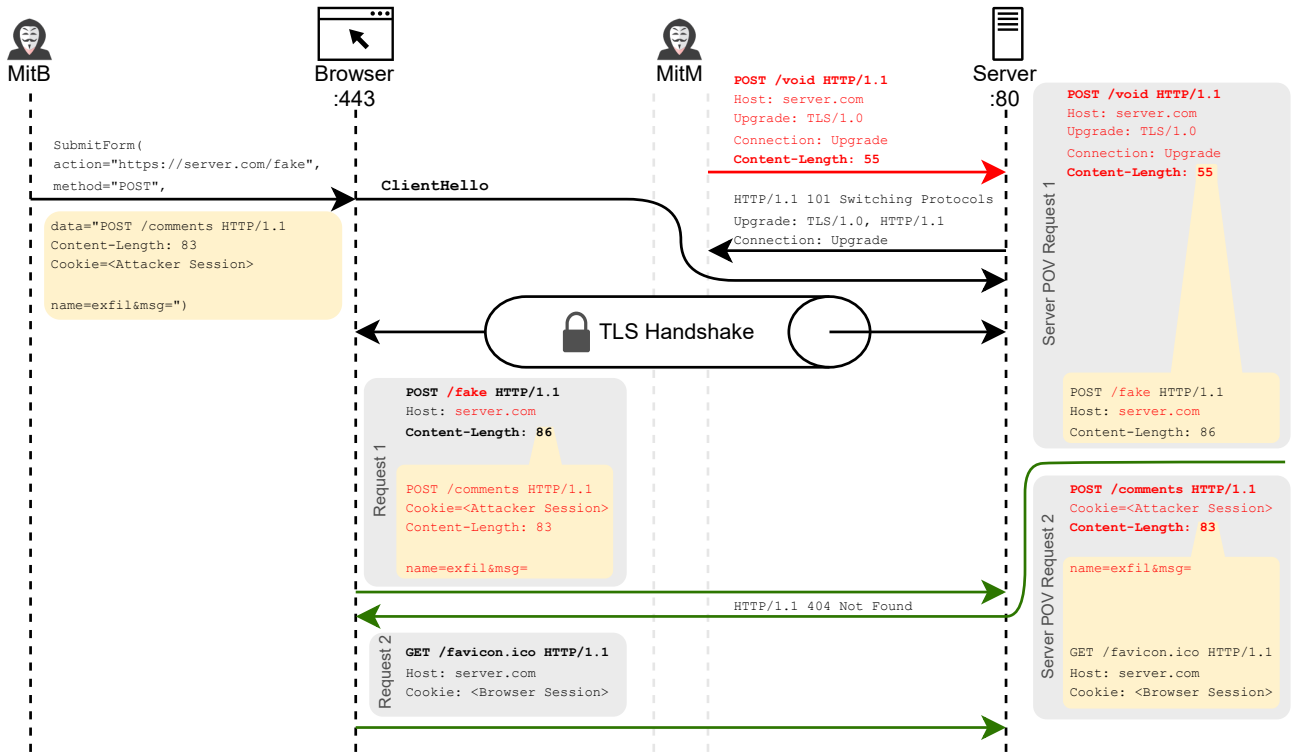

Figure 5: Sketch of an exploit for the HTTP POST behavior of Apache/2.4.62 (without padding). When upgrading a plain connection to TLS, Apache expects the request body to be sent through TLS. An attacker can abuse this, causing the request (Request 1) of a victim to be misinterpreted as a request body. In the MitB/MitM-attacker model, an attacker can further abuse this to prefix a second request (Request 2) with attacker-controlled data. This enables an attacker to use all cookies a victim sends (including HttpOnly and Secure) as HTTP post data, possibly disclosing them to the attacker (e.g., via posting a comment). We highlight attacker controllable request fields in red and lines defining the request (method, path, length) in **bold**. Green arrows denote messages sent and protected via TLS.

length. Icecast and Cyrus IMAP only finish the upgrade request after receiving enough POST data.

PoC – Apache Request Body Desynchronization Leaks Cookie We implemented this attack as a PoC, as seen in Figure 5. In this PoC, we assume an HTTP application that uses cookies with the `HttpOnly` and `secure` flags enabled. These flags prevent an attacker from accessing the cookies via JavaScript or through insecure connections. Additionally, we assume that the web application provides a feature that allows users to submit POST data to a location accessible by the attacker. For this purpose, we implemented a basic product comment feature for our PoC.

The attack starts by visiting an attacker-controlled website, unrelated (cross-site) to the server we are attacking. The website issues a form POST request (Request 1 in Figure 5) over HTTPS to the target server. At this point, the MitM attacker performs our Opossum attack but specifies a content length in the upgrade request. Following the TLS upgrade, the Apache server misinterprets the HTTP body of Request 1 as POST data for the upgrade request, resulting in desynchronization. The POST body of Request 1 is now treated as a request. Since the attacker can freely control any headers in this request, they can set an overlong content length to exfiltrate the next request issued by the browser. In our experiments, we found that both Chrome and Firefox will reuse the connection for any subsequent subresource requests (e.g., favicons, scripts, images), triggered by the response of the upgrade request. In our PoC, the browser automatically issues a favicon request, which, because it is a same-origin request, carries all cookies, regardless of modern cookie restrictions. Because of the desynchronization, the raw favicon request is posted as a comment with the attacker's session (Server POV Request 2 in Figure 5), which leaks the cookies.

In our PoC we had to add padding to Request 1, to fill one entire TLS record (2^{14} bytes), so that the first record sent by the browser contains a complete request. In the MitM attacker script, we then split and delay every TLS record received, which allows Apache to respond to the first request. Otherwise, Request 1 would not finish successfully in the browser,

and the connection could not be reused. For simplicity, we do not show this in Figure 5.

5 Opossum Attack on Protocols Beyond HTTP

As mentioned in Section 3, the Opossum attack affects any synchronous protocol that uses opportunistic and implicit TLS simultaneously and differs in the application protocol after the TLS handshake. In the protocols we explore in this paper, the observed difference is the communication order after the TLS handshake. If the peer who sends the first message after the handshake changes in the two variants, we consider the protocol to be affected by the Opossum attack. Due to the large number of TLS-based protocols, we do not provide a detailed exploitability analysis of all affected protocols. However, we believe the impact on these protocols is likely much lower than on HTTP. This is partly due to the MitB attacker model, which is usually not available in other protocols, but also due to how opportunistic TLS was integrated into HTTP. In HTTP, the attacker can send a *custom request*, and the victim client receives the response to that request as their response. In many other protocols, the attacker can merely desynchronize with a message they have little control over (e.g., a banner message), which makes chaining a full exploit together hard, if not impossible. While we cannot show a practical exploit beyond a Proof-of-Concept for desynchronization (see Appendix B), affected protocols give a Man-in-the-Middle attacker more capabilities than expected to influence the exchanged messages (and their interpretation). Table 1 summarizes the protocols we investigated in this work. In addition to HTTP, we investigated 14 protocols that support opportunistic TLS. Our selection includes common email and messaging protocols, protocols used for database access, and those for remote access or file transfer. For each of these protocols, we evaluated their support for implicit TLS and whether it is susceptible to the Opossum attack.

5.1 Additionally Affected Protocols

FTP The File Transport Protocol supports implicit and opportunistic TLS (via STARTTLS) [35]. While implicit TLS was never officially specified for FTP, it is still commonly implemented. For implicit TLS, the server is the first to send a message after the TLS handshake, while for opportunistic TLS, the client is the first to send a message [35]. When performing the Opossum attack O2I, the client will receive the FTP banner as the response to its first request after the TLS handshake. The FTP banner contains a `220` response code. How the client reacts to this is application-specific. If the client accepts this response code as a valid response to its first message after the TLS handshake, the desynchronization may continue. We visualize the behavior in Figure 6.

Protocol	TLS: Implicit	TLS: Opportunistic	Impact	
FTP	$\xleftarrow{S}$	$\xrightarrow{C}$	!	O2I
HTTP 1.1	$\xrightarrow{C}$	$\xleftarrow{S}$	ϟ	I2O
IMAP	$\xleftarrow{S}$	$\xrightarrow{C}$	(!)	O2I
IRC	$\xrightarrow{C}$	$\xrightarrow{C}$		
LDAP	$\xrightarrow{C}$	$\xrightarrow{C}$		
LMTP	$\xleftarrow{S}$	$\xrightarrow{C}$	!	O2I
Managed Sieve	$\xleftarrow{S}$	$\xleftarrow{S}$		
NNTP	$\xleftarrow{S}$	$\xrightarrow{C}$	!	O2I
POP3	$\xleftarrow{S}$	$\xrightarrow{C}$	!	O2I
PostgreSQL	$\xrightarrow{C}$	$\xrightarrow{C}$		
RDP	$\xrightarrow{C}$	$\xrightarrow{C}$		
SMTP	$\xleftarrow{S}$	$\xrightarrow{C}$	!	O2I
MySQL		$\xrightarrow{C}$		
NBD		$\xrightarrow{C}$		
XMPP		$\xrightarrow{C}$		

! Affected ϟ Exploit shown

$\xrightarrow{C}$ First AppData from client $\xleftarrow{S}$ First AppData from server

O2I Opportunistic to implicit scenario

I2O Implicit to opportunistic scenario

Table 1: Summary of our evaluation of various protocols in the context of the Opossum attack. The last column indicates if a protocol is susceptible or vulnerable to an attack scenario. To be susceptible, protocols need to offer both implicit and opportunistic TLS, and the direction of the first application data (sent by client or by server) must differ between the two modes. IMAP is only theoretically affected, as IMAP is not a synchronized protocol, it cannot be desynchronized.

IMAP In IMAP, both opportunistic and implicit TLS are supported [40, 55]. However, in contrast to all other analyzed protocols, IMAP is not a synchronous protocol. Instead, request-response pairs are matched by a tag chosen by the client. It is still possible to mix implicit and opportunistic TLS (O2I) to perform a 'prefix' injection into the channel, which should not be possible in a secure channel. However, the prefix that can be injected is only the Banner, which is an *untagged* response. We thus conclude that a desynchronization with the Opossum attack is not possible.

SMTP The Simple Mail Transfer Protocol supports opportunistic and implicit TLS [40, 52]. After the handshake in an implicit TLS connection, the SMTP server sends a banner with a `220` status code, while in an opportunistic TLS connection, the server is waiting for a command from the client, which makes SMTP vulnerable in the O2I scenario. This allows an attacker to forward an opportunistic client to

an implicit server, allowing the attacker to forward the server banner in response to the first request.

POP3 Similarly, the Post Office Protocol 3 supports opportunistic and implicit TLS [40, 55]. Regarding the Opossum attack, POP3 behaves analogously to SMTP (O2I).

LMTP The Local Mail Transfer Protocol is a mail transfer protocol similar to SMTP but designed for local message delivery. Regarding the Opossum attack, LMTP behaves analogously to SMTP (O2I).

NNTP The Network News Transfer Protocol is a protocol that transfers Usenet messages from a client to a server or between servers. It supports both opportunistic and implicit TLS [29, 53]. Regarding the Opossum attack, NNTP behaves analogously to SMTP (O2I).

5.2 Protocols Not Affected

The following protocols are not affected by a desynchronization of the Opossum attack because they only support opportunistic TLS, or they behave equally on the application layer when using implicit and opportunistic TLS.

Tunneled Protocols It is possible to create scenarios where implicit TLS is transparently added to protocols that otherwise do not support it by adding a transparent TLS proxy in front of them. This could make protocols that only support opportunistic TLS also vulnerable to the Opossum attack if they are deployed alongside a tunneled implicit variant.

MySQL MySQL is an open-source relational database management system. It can be configured to use TLS and supports opportunistic TLS. However, it is impossible to configure it to use implicit TLS, preventing the confusion. If implicit TLS is provided with TLS proxies, confusion would be possible. An attacker could forward an opportunistic client to an implicit server, allowing the attacker to forward the Server Greeting in response to the first request.

NBD The Network Block Device protocol enables remote access to block storage devices, such as hard disks. Like MySQL, it supports opportunistic TLS but does not support implicit TLS, which prevents confusion. If implicit TLS is provided with TLS proxies, confusion would be possible. An attacker could forward an opportunistic client to an implicit server, allowing the attacker to forward the server's initial NBD handshake message in response to the first request.

PostgreSQL Analog to MySQL, PostgreSQL is also an open-source relational database management system. It can be configured to use TLS and supports implicit and opportunistic TLS. Before and after the TLS handshake, the client is always expected to send the first message, preventing a desynchronization.

LDAP The Lightweight Directory Access Protocol is a protocol for interacting with distributed directory services. It supports both implicit and opportunistic TLS [70]. Analog to PostgreSQL, before and after the TLS handshake, the client is always expected to send the first message, preventing desynchronization.

RDP The Remote Desktop Protocol is a proprietary protocol developed by Microsoft that lets users access another computer over a network connection. It supports both implicit TLS and opportunistic TLS. Analog to PostgreSQL, before and after the TLS handshake, the client is always expected to send the first message, preventing desynchronization.

IRC The Internet Relay Chat is a text-based communication protocol that supports implicit TLS and opportunistic TLS. However, the IRCv3 Working Group has deprecated opportunistic TLS to encourage IRC networks to adopt implicit TLS exclusively. Analog to PostgreSQL, before and after the TLS handshake, the client is always expected to send the first message, preventing desynchronization.

Managed Sieve The Managed Sieve protocol is a protocol that manages email filtering rules remotely. It supports both opportunistic and implicit TLS [50]. Before and after the TLS handshake, the server always sends the first message, preventing a desynchronization.

XMPP The Extensible Messaging and Presence Protocol is an XML-based communication protocol that supports opportunistic TLS but does not support implicit TLS, which prevents confusion.

6 Large-Scale Internet Study

To assess the prevalence of susceptibility to the Opossum attack, we conducted Internet-wide scans. Since it is unclear where HTTP with opportunistic TLS is deployed, we scanned all ports for which the Shodan[8] search engine returned at least one million hits for "HTTP". In total, Shodan identified 31 ports with over one million hits. Additionally, based on specific RFCs [30, 57, 73], we included four more ports where HTTP servers with opportunistic TLS might be present.

[8] https://www.shodan.io

For each port, we first performed a TCP-SYN scan using ZMap [28]. Then we scanned the resulting hosts with ZGrab2 [27] by sending an HTTP request that attempts to upgrade the connection to TLS using the `Upgrade` header. Since we expected negotiation issues, we tested each TLS version individually by sending separate HTTP requests for each TLS version in the `Upgrade` header.

6.1 HTTP Results

In total, we evaluated 327,767,219 servers distributed over 35 ports. Among these, 36,788 replied with an HTTP 101 code indicating support for our requested upgrade and 33,274 specifically signaled TLS support through the `Upgrade` header field. We analyzed all hosts that initially responded with an HTTP 101 code and found that 20,121 responded to TLS handshake messages sent after the upgrade request. Below, we provide further details on the response patterns we observed in our additional scans. A full overview of the results for each port is shown in Table 2.

Associated HTTPS Servers To find a lower bound for the impact of the desynchronization attack, we extracted domain names from received X.509 certificates and tested the domains for HTTPS support on port 443. While most servers provided certificates scoped to local domains or preset certificates without any listed domains, we could extract 5,872 valid domain names. We found that 2,268 (38.6%) of these domains also support implicit TLS on port 443 and 539 (9.2%) protect HTTPS traffic with a certificate that is also valid for the web server offering opportunistic TLS, which ultimately enables the Opossum attack.

Unexpected SSH Responses to TLS Upgrade Requests During our scan, we observed that not all hosts that sent an HTTP *101 Switching Protocols* response proceeded with a TLS handshake. Instead, 2,556 servers (6.9%) responded with either an SSH banner or an SSH Key Exchange Init message.

Reflected Upgrade Headers On all ports except for 80 and 631, most servers did not proceed with the TLS handshake, despite signaling TLS support in the 101 HTTP response. We further tested these servers and found that the majority echo the content of our `Upgrade` header and immediately close the connection after sending the 101 response. We assume these are WebSocket servers because sending a request to upgrade HTTP to WebSocket, rather than TLS, yielded additional WebSocket-related headers in the 101 responses and kept the TCP connection alive.

TLS Upgrades Without Upgrade Header 3,477 servers supported TLS even though their HTTP 101 response did not include the `Upgrade` header indicating any TLS version.

Port	IPs[1]	HTTP Upgrade[2]	TLS Upgrade[3]	Speaks TLS[4]
80	54,546,240	4,168	3,185	2,399
443	53,373,461	567	472	46
7547	38,621,425	388	388	0
8443	10,338,350	454	441	11
8089	9,930,896	438	419	4
8080	9,894,096	1,665	1,146	753
21	9,766,232	307	307	0
8085	9,689,861	416	402	12
7170	8,360,836	189	189	0
4567	8,189,361	403	403	1
8000	7,577,744	512	512	80
8008	6,803,053	387	387	16
9000	6,399,101	303	295	3
8081	5,454,634	465	446	49
2082	5,244,141	774	308	0
2087	5,211,866	281	281	0
2083	5,191,865	294	291	0
2086	5,128,245	829	252	0
8888	4,902,448	267	260	10
1024	4,649,290	218	218	2
8880	4,462,136	594	176	3
9080	4,428,670	309	298	58
5985	4,417,712	227	227	0
3000	4,329,660	262	262	1
5001	4,227,576	256	256	3
8001	4,223,223	348	336	14
5000	4,123,893	209	209	2
81	4,119,701	238	238	21
8090	4,047,896	348	336	14
3128	3,877,703	246	246	0
7777	3,835,388	307	307	0
9100	3,366,261	532	525	6
631	3,218,160	19,006	18,666	16,618
1344	3,154,290	300	300	1
702	2,661,805	314	314	1
$\sum$	327,767,219	36,788	33,274	20,121

1: Unique IPs that responded successfully to a TCP SYN.

2: Unique IPs that responded with an HTTP 101.

3: Unique IPs that responded with an HTTP 101 and an Upgrade header for TLS.

4: Unique IPs that responded with a ServerHello / Alert to our ClientHello.

Table 2: Results of our scan for support of opportunistic HTTP. We found 36,788 servers that advertise support for opportunistic TLS, with 20,121 showing evidence that they actually support it.

Observed Server Headers During our analysis, we also examined the `Server` headers returned by hosts that support the TLS upgrade. If present, this header indicates the server software used and the version used. Most hosts did not send a `Server` header. Among those that did, the headers contained different versions of Apache, CUPS, IPP, PAPPL, and Icecast.

	TLS		
Protocol	**Implicit**	**Opportunistic**	**Susceptible**
FTP[1]	115,799	2,083,439	219,200
IMAP[2]	1,778,397	1,679,746	1,473,974
POP3[3]	1,493,423	1,327,032	1,139,443
SMTP[4]	969,709	420,927	243,153

1: Scanned port: 990 and 21
2: Scanned port: 993 and 143
3: Scanned port: 995 and 110
4: Scanned port: 465 and 587

Table 3: Results of our scan for implicit and opportunistic TLS support in FTP, IMAP, POP3, and SMTP. The last column shows that many of these hosts are susceptible to the Opossum attack.

6.2 Results for FTP, IMAP, POP3, and SMTP

In addition to HTTP, we also examined a set of susceptible protocols, including FTP, IMAP, POP3, and SMTP. For these (currently) non-exploitable protocols, we used the same approach as in our HTTP study. For each protocol, we scanned the ports used for implicit TLS and those used for opportunistic TLS. First, we performed a TCP-SYN scan with ZMap to identify hosts responding to the relevant ports. Then, we used ZGrab2 with the existing modules to check for implicit and opportunistic TLS support. Table 3 summarizes the results.

For FTP, 2M servers support opportunistic TLS, while only about 115k support implicit TLS. For both IMAP and POP3, the number of servers supporting implicit and opportunistic TLS was similar. In contrast, for SMTP, we found twice as many servers with implicit TLS support as with opportunistic TLS.

We also evaluated how many hosts support implicit and opportunistic TLS and present a certificate with the same hostname. This allowed us to assess the relevance of the Opossum attack for each protocol. Our analysis shows that most IMAP and POP3 servers are susceptible to the Opossum attack, with less prevalence in SMTP. For FTP, we see that opportunistic TLS is dominant, with the lack of implicit TLS preventing the attack. In FTP, only the opportunistic TLS variant is officially standardized. While the implicit variant is considered more secure, there is no official RFC for implicit TLS for FTP, which may explain the observed difference.

7 Mitigation

We attribute the core reason for the Opossum attack to the authentication flaw from the ALPACA attack [18].

For the Opossum attack, strict ALPN verification does not work, as both protocols (implicit and opportunistic TLS) are using the *same* ALPN identifier. However, in affected protocols, from a theoretical modeling perspective, the protocol after an opportunistic upgrade and the protocol after an implicit TLS are two different application layer protocols. The two protocols are identical but shifted by the initial message, which creates the desynchronization.

It is possible to fix the Opossum attack similarly to the ALPACA attack [18]. By introducing a *special* ALPN string for the opportunistic protocol, implementing it into respective clients, and enforcing it on the server side, one could prevent the attack, as the server could then distinguish opportunistic TLS from implicit TLS. However, adding additional TLS extensions or ALPN strings to already deployed implementations is a slow process, and it is unlikely that such patches will reach implementations in significant numbers.

A broader fix for our attacks requires coordination between implementations to fix the issue, as it exploits standard-compliant behavior. Opportunistic TLS is known to weaken the security of email protocols [59]. With the Opossum attack, we can additionally show that many protocols using opportunistic TLS are suffering from the Opossum attack. Our preferred solution to the Opossum attack is the deprecation of *all opportunistic TLS protocols*. Upgrading software to implicit TLS is typically easy for developers to do and is, for the most part, application-independent. However, this requires that some protocols need to standardize implicit TLS, including protocols like FTP, MySQL, NBD, and XMPP, which may face resistance.

Regarding HTTP, we present practical exploits, but found only a small number of potentially affected servers in the wild. Since HTTP-to-TLS upgrades are an esoteric feature that does not really have a place in the modern HTTP ecosystem, we think that deprecating the feature is the best way forward. It is unlikely that other countermeasures will reach broad adoption, as proper countermeasures require integration on both ends.

8 Related Work

A pillar in opportunistic encryption is RFC 7435 [24], which introduces the principles of Opportunistic Security design. The document gives insights and guidance on the use of opportunistic encryption and authentication. The document is related to DANE [25], which defines an (opportunistic) DNS-based alternative to traditional PKI. An oversight in opportunistic security standards is that the security considerations did not account for attacks where opportunistic security would compromise implicit variants, as exploited in our attack.

Poddebniak et al. [59] analyzed the security of STARTTLS implementations in the context of SMTP, POP3 and IMAP, and found plaintext injection attacks caused by bugs in the respective implementations. Using these bugs in opportunistic TLS, Poddebniak et al. then presented an attack where they used the plaintext injection vulnerabilities to attack implicit TLS clients (I2O), showing that simply providing opportunistic TLS alongside implicit TLS can reduce the security of the implicit variant.

The Opossum exploit on HTTP is related to the TLS Renegotiation attack [62], which allows for arbitrary prefix injec-

tion. In contrast to the Renegotiation attack, the Opossum attack does not allow for an arbitrary prefix but only allows for the injection of full HTTP requests. The Renegotiation attack was mitigated by the introduction of the *Renegotiation Indication* extension [66].

In the context of HTTP, desynchronization attacks by request smuggling have been studied by James Kettle [43–46]. *HTTP request smuggling* exploits inconsistencies in how front-end and back-end systems process HTTP headers, particularly `Content-Length` and `Transfer-Encoding`. By crafting an ambiguous request, an attacker can desynchronize the front-end and back-end servers, enabling them to "smuggle" malicious data.

Canetti and Krawczyk have formally analyzed channel security of key exchange protocols [19]. For the TLS record layer, Paterson et al. [58] define stateful length-hiding authenticated encryption (sLHAE). Based on this, [42, 49] analyze the TLS handshake and record layer using the framework of authenticated and confidential channel establishment (ACCE). A more general approach is given by Fischlin et al. [34], who define plaintext integrity (INT-PST) for arbitrary data streams. Our work shows that HTTPS Upgrade does not provide integrity of plaintext (INT-PST) as defined by Fischlin et al. [34].

9 Conclusions

In this work, we have shown that deploying opportunistic TLS alongside implicit TLS can cause desynchronization vulnerabilities in many application protocols, including HTTP 1.1, FTP, IMAP, SMTP, LMTP, NNTP, and POP3. We then showed how this desynchronization vulnerability, in the case of HTTP 1.1, can lead to full exploits, even if the feature was never implemented in modern browsers. While we did not show practical exploits for other vulnerable application layer protocols, the Opossum attack gives attackers capabilities that subvert the expectations of the developers on the TLS channel, which attackers can potentially exploit on a case-by-case basis. Our Internet scan shows that, while only a few HTTPS servers are potentially affected, many email and FTP servers are susceptible to the general Opossum attack. While the potential for exploits of non-HTTP protocols is low, it gives attackers an unexpected technique that might lead to implementation-specific exploits. The deployment of opportunistic security measures should *never* interfere with the strict version of the protocols. We, therefore, argue that it is time for the community to take a proactive position and deprecate opportunistic TLS in *all protocols* and to consider its deployment as harmful, independent of the presence of publicly known exploits for the concrete implementation and protocol.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This research was partially supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA – 390781972 and by the German Federal Ministry of Research, Technology and Space (BMFTR) through the project KoTeBi. Lukas Knittel and Sven Hebrok were supported by the research project "North-Rhine Westphalian Experts in Research on Digitalization (NERD II)", sponsored by the state of North Rhine-Westphalia – NERD II 005-2201-0014.

Open Science

As part of our artifacts, we release our PoC exploits for HTTPS and other protocols we identified as susceptible (see Table 1). We also provide Dockerfiles for software supporting opportunistic HTTP (see Section 4.2). All artifacts can be found at `https://doi.org/10.5281/zenodo.17855525`.

Ethical Considerations

Stakeholders For each step of our methodology, we considered potentially affected parties. The evaluation of protocols affects standardization bodies as changes may be necessary to mitigate threats. The evaluation of implementations affects vendors as their products may become subject to attacks. The Internet-wide scan affects operators and organizations deploying server software as they might have to update their software or adapt their configuration to disable opportunistic TLS, particularly in HTTP. We further considered the broader impact on the general public, in particular users of affected services whose security and privacy depend on robust standards and implementations.

Scans As part of our evaluation, we conducted Internet scans to estimate the attack surface of our identified vulnerabilities. For our HTTP landscape analysis, we sent HTTP requests to web servers on various ports (see Table 2). While these connections may appear as unwanted noise to the respective server administrators, we only used standard HTTP features with well-formed messages and did not attempt to perform any exploits. The extent of our scans per host was further limited to few connections. Since we also searched for web servers deployed on ports beyond well-known HTTP(S) ports, we may have sent HTTP requests to non-HTTP endpoints, possibly causing parser errors. However, compared to the malformed traffic any endpoint exposed to the Internet faces, we do not expect these cases to be particularly critical. In our scans for FTP, IMAP, POP3, and SMTP we likewise only used standard, well-formed protocol messages for the

respective implicit and explicit TLS endpoints. We hence do not expect that our scans disrupted any services.

For all scans, we further respected the rules for Internet-wide scanning proposed by Durumeric et al. [28]. To this end, we established rDNS entries and a website that indicated the benign nature of our scans and offered the possibility of opting out of our study. Due to the limited scope of our scans, we do not expect that other components of the Internet infrastructure, such as routers along the path to the target web server, were affected to any measurable extent.

Identified Vulnerabilities Due to the sheer number of vulnerable hosts, we responsibly disclosed our findings to our local CERT. We additionally approached the IETF and advised them to deprecate the affected standards. Finally, we contacted popular software products for which we could identify concrete exploits and advised them to remove the feature. In response to our disclosure, Cyrus IMAP changed its defaults to implicit TLS and Apache2 removed support for RFC 2817.

Decision to Conduct and Publish our Research We chose to conduct this research to identify and help mitigate flaws in Internet standards and implementations that could (potentially) already be exploited by adversaries. To minimize harm and enable timely remediation, we followed a coordinated vulnerability disclosure process: we first shared our findings with stakeholders able to implement fixes at the standard, implementation, and deployment levels, and only then disclosed our results to the wider public. We believe that this process, and the resulting improvements to the ecosystem, justify both the study and its publication.

References

[1] The apache http server project. https://httpd.apache.org/. Accessed: 2025-02-14.

[2] Apache james. https://james.apache.org/. Accessed: 2025-02-14.

[3] Cyrus imap. https://github.com/cyrusimap/cyrus-imapd. Accessed: 2025-02-14.

[4] Dovecot. https://www.dovecot.org/. Accessed: 2025-02-14.

[5] Ghostscript printer application. https://github.com/OpenPrinting/ghostscript-printer-app. Accessed: 2025-02-14.

[6] Gutenprint printer application. https://github.com/OpenPrinting/gutenprint-printer-app. Accessed: 2025-02-14.

[7] Hp printer application. https://github.com/michaelrsweet/hp-printer-app. Accessed: 2025-02-14.

[8] Hplip printer application. https://github.com/OpenPrinting/hplip-printer-app. Accessed: 2025-02-14.

[9] Icecast-server. https://github.com/xiph/Icecast-Server. Accessed: 2025-02-14.

[10] Libshout. https://github.com/xiph/Icecast-libshout. Accessed: 2025-02-14.

[11] Lprint - a label printer application. https://github.com/michaelrsweet/lprint. Accessed: 2025-02-14.

[12] Mysql. https://www.mysql.com/. Accessed: 2025-02-03.

[13] Pappl - printer application framework. https://github.com/michaelrsweet/pappl. Accessed: 2025-02-14.

[14] Postgresql. https://www.postgresql.org/. Accessed: 2025-02-03.

[15] Postscript printer application. https://github.com/OpenPrinting/ps-printer-app. Accessed: 2025-02-14.

[16] Proftpd. http://www.proftpd.org/. Accessed: 2025-02-14.

[17] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society Press, May 2014.

[18] Marcus Brinkmann, Christian Dresen, Robert Merget, Damian Poddebniak, Jens Müller, Juraj Somorovsky, Jörg Schwenk, and Sebastian Schinzel. ALPACA: Application layer protocol confusion - analyzing and mitigating cracks in TLS authentication. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 4293–4310. USENIX Association, August 2021.

[19] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474. Springer, Berlin, Heidelberg, May 2001.

[20] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard), March 2003. Obsoleted by RFC 9051, updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858, 7817, 8314, 8437, 8474, 8996.

[21] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Historic), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465, 7507, 7919.

[22] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Historic), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176, 7465, 7507, 7919.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447, 9155.

[24] V. Dukhovni. Opportunistic Security: Some Protection Most of the Time. RFC 7435 (Informational), December 2014.

[25] V. Dukhovni and W. Hardaker. The DNS-Based Authentication of Named Entities (DANE) Protocol: Updates and Operational Guidance. RFC 7671 (Proposed Standard), October 2015.

[26] Thai Duong and Juliano Rizzo. Here come the $\oplus$ Ninjas. In *EKOparty*, 2011.

[27] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 542–553. ACM Press, October 2015.

[28] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In Samuel T. King, editor, *USENIX Security 2013*, pages 605–620. USENIX Association, August 2013.

[29] J. Elie. Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 8143 (Proposed Standard), April 2017.

[30] J. Elson and A. Cerpa. Internet Content Adaptation Protocol (ICAP). RFC 3507 (Informational), April 2003.

[31] C. Feather. Network News Transfer Protocol (NNTP). RFC 3977 (Proposed Standard), October 2006. Updated by RFC 6048.

[32] R. Fielding (Ed.), M. Nottingham (Ed.), and J. Reschke (Ed.). HTTP Semantics. RFC 9110 (Internet Standard), June 2022.

[33] R. Fielding (Ed.) and J. Reschke (Ed.). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231 (Proposed Standard), June 2014. Obsoleted by RFC 9110.

[34] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. Data is a stream: Security of stream-based channels. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564. Springer, Berlin, Heidelberg, August 2015.

[35] P. Ford-Hutchinson. Securing FTP with TLS. RFC 4217 (Proposed Standard), October 2005. Updated by RFC 8996.

[36] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[37] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard), July 2014. Updated by RFC 8447.

[38] Mario Heiderich, Christopher Späth, and Jörg Schwenk. Dompurify: Client-side protection against xss and markup injection. pages 116–134, 08 2017.

[39] K. Hickman. The SSL Protocol. Internet Draft, http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt, April 1995.

[40] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), February 2002. Updated by RFC 7817.

[41] T. Huth, J. Freimann, V. Zimmer, and D. Thaler. DHCPv6 Options for Network Boot. RFC 5970 (Proposed Standard), September 2010.

[42] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Berlin, Heidelberg, August 2012.

[43] James Kettle. Breaking the chains on HTTP Request Smuggler. PortSwigger Research Articles. https://portswigger.net/research/breaking-the-chains-on-http-request-smuggler.

[44] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Research Articles. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[45] James Kettle. HTTP Desync Attacks: what happened next. PortSwigger Research Articles. https://portswigger.net/research/http-desync-attacks-what-happened-next.

[46] James Kettle. HTTP/2: The Sequel is Always Worse. PortSwigger Research Articles. https://portswigger.net/research/http2.

[47] R. Khare and S. Lawrence. Upgrading to TLS Within HTTP/1.1. RFC 2817 (Proposed Standard), May 2000. Updated by RFCs 7230, 7231.

[48] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), October 2008. Updated by RFC 7504.

[49] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Berlin, Heidelberg, August 2013.

[50] A. Melnikov (Ed.) and T. Martin. A Protocol for Remotely Managing Sieve Scripts. RFC 5804 (Proposed Standard), July 2010. Updated by RFCs 7817, 8553.

[51] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. *Security Advisory*, 21:34–58, 2014.

[52] K. Moore and C. Newman. Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access. RFC 8314 (Proposed Standard), January 2018. Updated by RFC 8997.

[53] K. Murchison, J. Vinocur, and C. Newman. Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 4642 (Proposed Standard), October 2006. Updated by RFCs 8143, 8996.

[54] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939 (Internet Standard), May 1996. Updated by RFCs 1957, 2449, 6186, 8314.

[55] C. Newman. Using TLS with IMAP, POP3 and ACAP. RFC 2595 (Proposed Standard), June 1999. Updated by RFCs 4616, 7817, 8314.

[56] C. Newman. ESMTP and LMTP Transmission Types Registration. RFC 3848 (Draft Standard), July 2004.

[57] A. Newton and M. Sanz. Using the Internet Registry Information Service (IRIS) over the Blocks Extensible Exchange Protocol (BEEP). RFC 3983 (Proposed Standard), January 2005. Updated by RFC 8996.

[58] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Berlin, Heidelberg, December 2011.

[59] Damian Poddebniak, Fabian Ising, Hanno Böck, and Sebastian Schinzel. Why TLS is better without STARTTLS: A security analysis of STARTTLS in the email context. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 4365–4382. USENIX Association, August 2021.

[60] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Internet Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151.

[61] Network Block Device Project. The nbd protocol. https://github.com/NetworkBlockDevice/nbd/blob/master/doc/proto.md. Accessed: 2025-02-03.

[62] M. Ray and S. Dispensa. Authentication gap in TLS renegotiation, 2009.

[63] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Obsoleted by RFC 9110, updated by RFCs 5785, 7230.

[64] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[65] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[66] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010.

[67] Juliano Rizzo and Thai Duong. The CRIME attack. In *Ekoparty Security Conference 2012*, volume 2012, 2012.

[68] J. Rosenberg. The Extensible Markup Language (XML) Configuration Access Protocol (XCAP). RFC 4825 (Proposed Standard), May 2007.

[69] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. Updated by RFCs 7590, 8553.

[70] J. Sermersheim (Ed.). Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511 (Proposed Standard), June 2006.

[71] Y. Sheffer, P. Saint-Andre, and T. Fossati. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 9325 (Best Current Practice), November 2022.

[72] Ben Smyth and Alfredo Pironti. Truncating TLS connections to violate beliefs in web applications. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.

[73] M. Sweet and I. McDonald. Internet Printing Protocol/1.1: Encoding and Transport. RFC 8010 (Internet Standard), January 2017.

[74] Angelo Prado Yoel Gluck, Neal Harris. Breach: Reviving the crime attack, July 2013.

A Opossum Attack on FTPS

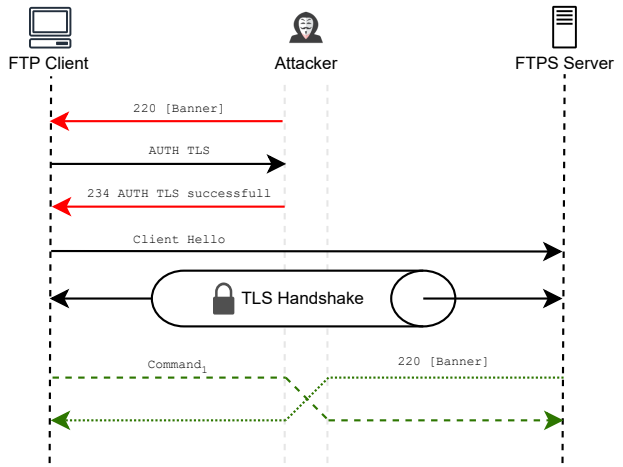

Figure 6: A sketch of the Opossum attack on FTPS. Different line types denote unidirectional TLS channels.

B Proof-of-Concept Evaluation

We validated the descriptions of both implicit and opportunistic TLS for the susceptible protocols by deploying a client and server for each protocol locally. Specifically, for SMTP, POP3, and IMAP, we used an Apache James [2] server. For LMTP, we used a Dovecot [4] server, and for FTP a ProFTPD [16] server. On the client side, we implemented a Python script that establishes a connection to a server using the official Python module for that protocol, confirming the described desynchronization.