2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA) | 979-8-3315-5383-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/ETFA65518.2025.11205612

Towards Privacy Analysis of Internet-based Messaging Applications

Muqaddas Naz
Centre for Sustainable Cyber Security
University of Greenwich, London
Email: Muqaddas.Naz@greenwich.ac.uk

Muhammad Taimoor Khan
Centre for Sustainable Cyber Security
University of Greenwich, London
Email: m.khan@greenwich.ac.uk

***Abstract*—Privacy of Internet-based messaging applications involves establishing multiple requirements such as confidentiality, anonymity, pseudonymity, and consent, each typically addressed in isolation using different techniques. To the best of our knowledge, no unified framework supports the privacy analysis of these aspects together. Therefore, the main goal of our work is to detect privacy breaches based on a unified modelling of the relationship among the privacy requirements by static program analysis. In this paper, we present initial results on the privacy (i.e., confidentiality) analysis of Ejabberd server – an open-source XMPP backend server used by popular messaging applications like WhatsApp. Based on the confidentiality modelling, we perform a privacy analysis of key Ejabberd modules that are responsible for user authentication. The findings highlight the module implementations that involve potential privacy breaches and serve as a basis for our future work towards developing a unified privacy evaluation framework.**

I. Introduction

Data privacy in popular messaging apps such as WhatsApp, Signal, and Telegram, has become a major concern. These platforms handle sensitive content such as messages, credentials, and location data, making them prime targets for breaches. A 2022 Internet Society report showed that over 60% of users cited privacy breaches as a top concern, with 45% reducing usage or switching platforms after incidents [1]. IBM also reported that the average cost of a data breach in the communications sector reached $5.72 million per breach [2]. These figures underscore the urgent need for strong privacy protection in such applications and underlying systems. Ensuring privacy in messaging applications is a complex task due to the multifaceted nature of privacy itself. Key requirements such as confidentiality, anonymity, pseudonymity, unlinkability, consent management, identity control, and secrecy each address different aspects of user protection. However, implementing these requirements simultaneously presents significant challenges. For instance, maintaining strong confidentiality through encryption can conflict with the need for data auditing or moderation. Similarly, enabling user anonymity and unlinkability can complicate authentication and accountability mechanisms. These trade-offs are further complicated in distributed or federated systems where data flows across multiple components, often with varying levels of trust, regulation, and infrastructure security. As a result, ensuring consistent and comprehensive privacy across such platforms remains an ongoing challenge for developers and researchers alike. To address these challenges, various approaches have been proposed to detect privacy vulnerabilities in messaging systems. For instance, TaintDroid [3] enables dynamic taint tracking on Android to detect data leaks, while FlowDroid [4] provides static taint analysis for Android apps. Formal methods such as ProVerif [5] are used to verify security protocols. However, these tools often focus on specific platforms or isolated properties, lacking support for multi-dimensional privacy analysis or real-world server-side systems. STRIDE [6] and LINDDUN [7] offer structured frameworks while implementation-level privacy is missing. Meta's internal tool InferERL [8] reportedly offers static analysis for Erlang-based systems, but it is not publicly available, highlighting a lack of accessible tools for privacy evaluation in practical communication platforms.

All of the above-mentioned and contemporary approaches are limited to the detection of specific categories of privacy requirements and do not provide a comprehensive framework for the unified analysis of the fundamentally diverse nature of these requirements, which is the focus of our project. We aim to develop a unified framework capable of analyzing these multiple privacy requirements in software applications in general and in messaging applications in particular, which is the first such work to the best of our knowledge. However, this paper explores how privacy properties specifically confidentiality can be evaluated within real world messaging systems using program analysis techniques. In this paper, the terms "confidentiality" and "privacy" are used interchangeably. As a first step, we focus on the static confidentiality analysis of Ejabberd, an open-source XMPP-based messaging server widely used in communication networks. To conduct this investigation, we extended RefactorErl [9], an open-source static source code analysis and transformation tool for the Erlang programming language. RefactorErl enables detailed structural and semantic analysis of the Ejabberd codebase, allowing us to trace data flows and detect potential vulnerabilities. Furthermore, we have extended our privacy analysis by identifying other related security vulnerabilities in Ejabberd

using vulnerability checker provided by RefactorErl [10]. The rest of the paper is organized as follows: Section II presents the theoretical background, while Section III describes the methodology, implementation and identified confidentiality vulnerabilities, and their implications. Section IV reports on identifying other vulnerabilities that breach privacy through extended analysis of Ejabberd. Section V reviews related work, and Section VI concludes the paper with future directions.

II. Background

In this section, we introduce the main features of Ejabberd which is the main test of our methodology, followed by the modelling of confidentiality for privacy analysis of Ejabberd.

A. Ejabberd

Ejabberd is an open-source XMPP (Extensible Messaging and Presence Protocol) server known for its robustness, scalability, and real-time messaging capabilities. Developed in Erlang, Ejabberd is designed to handle large number of concurrent connections, making it a popular choice for large-scale messaging systems. Among its key features are support for XMPP extensions, load balancing, modular architecture, and strong fault-tolerance mechanisms. One of the most notable real-world applications of Ejabberd is its use in the early architecture of WhatsApp. WhatsApp leveraged Ejabberd to manage its messaging infrastructure due to its ability to support millions of simultaneous users and deliver real-time communication at scale. This highlights Ejabberd's suitability for high-performance, secure, and scalable messaging platforms. In the context of our study on confidentiality assurance in messaging systems, we have selected Ejabberd due to its modular architecture and widespread use. Our analysis focuses on key authentication and administrative modules: (i) `ejabberd_auth`, (ii) `ejabberd_auth_anonymous`, (iii) `ejabberd_auth_external`, (iv) `ejabberd_auth_mnesia`, (v) `ejabberd_auth_pam`, (vi) `mod_admin_extra`, and (vii) `ejabberd_auth_sql`. These modules handle critical operations such as standard and anonymous user authentication, external and database-backed credential verification, PAM-based system authentication, and administrative control. Their direct involvement in managing user identity and access makes them prime candidates for confidentiality analysis. Our goal is to assess their implementation against confidentiality breaches through model-based analysis.

B. Confidentiality

Typically, confidentiality is to protect sensitive data so that unauthorized parties cannot view the data [11], which is basis of our confidentiality model as shown in Figure 1 where comments are preceded by the # symbol.

Briefly in the model, U denote the set of all users of a program, and C the set of its internal components, such as methods, classes, or system elements such as file system or operating system calls. The union $U \cup C$ forms the set of parties P that may interact with sensitive variables during program execution. Among these, authorized parties (AP) are identified by an access control policy. To preserve confidentiality, we define an access control function *access*, ensuring that every access to a sensitive variable is performed solely by authorized users and components.

To analyze modelled behavior of confidentiality at runtime, we define an environment π as a mapping of program variables to their current values, i.e., $\pi : \text{Variable} \rightarrow \text{Value}$. Sensitive information is isolated using a filter function $data(\pi) = \{(I : V) \in \pi \mid V \text{ is private/sensitive}\}$. An access control list (ACL) maps which parties have what kind of access to which variables. The resulting access map ($\Gamma : \text{Variable} \rightarrow P \rightarrow \text{Access}$) models the effective access permissions to sensitive data.

The authorized access map for a given program state and ACL is computed as the union of authorized users and components for sensitive variables as modelled through *authorisedParties*. Each of these functions determines which trusted users and components have specific access (e.g., read*, write*) to sensitive variables. A violation of confidentiality occurs when a party not in this map tries to access or leak sensitive data through variable(s).

III. Confidentiality Analysis of Ejabberd

In this section, we discuss methodology and its implementation, and the results of Ejabberd privacy analysis.

A. Methodology and Implementation

To analyze privacy violations in general and confidentiality violations in particular within real-world Internet-based messaging applications, we developed a static analysis framework grounded in formal access control theory and implemented it over the Ejabberd server codebase. Figure 2 illustrates the overall workflow of our privacy analysis approach. We begin by modelling confidentiality (privacy) as a property of access control over sensitive variables. As defined in Section II-B, the confidentiality policy enforces that only authorized parties (i.e., users or components) identified through the defined access control function may access specific sensitive variables. Any unauthorized access to these variables constitutes a violation.

Based on the confidentiality model (Figure 1), we generate the following two privacy conditions to analyze confidentiality of Ejabberd:

1) when an un-trusted user appears to access sensitive information as modelled using *authorisedUsers*, and
2) when an un-trusted component appears to access sensitive information as modelled using *authorisedComponents*.

Due to access issues with the users of the application, we cannot test the former condition. Furthermore, to make the experiment standalone, we focused on analyzing the latter condition. Specifically, the condition serves as a probe into the behavior of the program, particularly targeting code paths and data flows that could potentially lead to breach

#The Environment maps variables to values

$$\text{Environment} = \pi = \text{Variable} \mapsto \text{Value}$$

#The AccessList lists parties, with their specific access to the variables

$$\text{AccessList} = ACL = \text{Parties} \mapsto \text{Variable} \mapsto \text{Access}$$

#The AccessMap maps variables to parties with specific access to the variables

$$\text{AccessMap} = \Gamma = \text{Variable} \mapsto \text{Parties} \mapsto \text{Access}$$

$$\text{Access} = A = \{\texttt{read*}, \texttt{write*}\}$$

$$\text{Users} = U = \textit{a set of all users}$$

$$\text{Components} = C = \textit{a set of all components in the program}$$

$$\text{Parties} = P = \{\forall\ u \in U \cup c \in C : \text{authorisedParties}(data(\pi), ACL, u \cup c)\}$$

#The data returns the Environment that contains pairs of only sensitive variables and their values

$$data : \text{Environment} \to \text{Environment}$$

$$data(\pi) = \{(I : V) \in \pi \mid V \textit{ is private/sensitive}\}$$

#The authorisedParties returns only those parties that have an access to variables

$$\text{authorisedParties} : \text{Environment} \times ACL \times P \to \Gamma$$

$$\begin{aligned}\text{authorisedParties}(data(\pi), ACL, P) = {} & \text{authorisedUsers}(data(\pi), ACL, users(P)) \\ & \cup^{+} \text{authorisedComponents}(data(\pi), ACL, components(P))\end{aligned}$$

#The authorisedUsers returns only those users that have an access to variables

$$\text{authorisedUsers} : \text{Environment} \times ACL \to \Gamma$$

$$\text{authorisedUsers}(data(\pi), ACL, P) = \left\{ \langle v : p : a \rangle \;\middle|\; \begin{array}{l} v \in vars(data(\pi)),\ p \in P,\ a \in A, \\ \text{elementOf}(\langle p : v : a \rangle, ACL) \end{array} \right\}$$

#The authorisedComponents returns only those components that have an access to variables

$$\text{authorisedComponents} : \text{Environment} \times ACL \to \Gamma$$

$$\text{authorisedComponents}(data(\pi), ACL, P) = \left\{ \langle v : p : a \rangle \;\middle|\; \begin{array}{l} v \in vars(data(\pi)),\ p \in P,\ a \in A, \\ \text{elementOf}(\langle p : v : a \rangle, ACL) \end{array} \right\}$$

Fig. 1: Confidentiality Model based on Access Control List

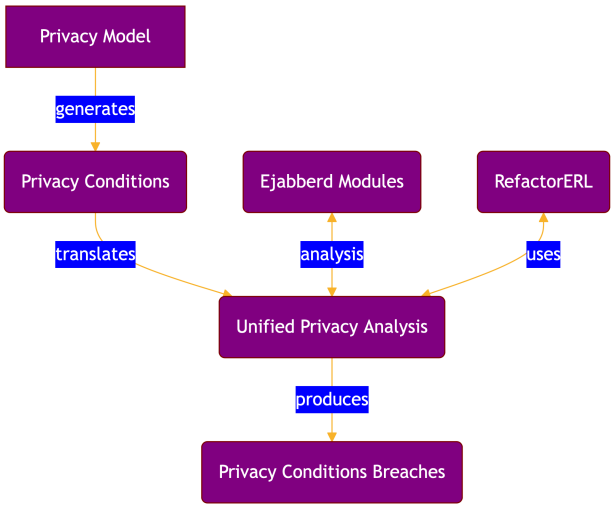

Fig. 2: Workflow of Privacy Analysis

of the condition through unauthorized data access. These conditions are translated into queries that are amenable for privacy analysis of the Ejabberd code.

Prior to privacy analysis, we identify Ejabberd modules where access control violations are most likely to occur. Since our confidentiality model is focused on authentication and access management, we concentrate our analysis of such modules which handles authentication and administrative components highlighted earlier in SectionII-A. As a next step, we analyze the Ejabberd codebase using a unified privacy analyzer built on top of RefactorErl [9], which is a static analysis tool for Erlang programs. The analyzer allows us to inspect the code structure, trace variable usage, and model data flows across the selected modules. Through this, we check whether any sensitive variable, as identified by the filter function in the environment state π, is accessed by components not included in the authorized access map Γ. This process implements our model within the context of real code. Finally, we check whether any unauthorized party accesses sensitive variables (as per our access control model). If such access is detected, it is flagged as a confidentiality breach. Otherwise, the module passes the confidentiality check. This framework not only

helps detect existing privacy flaws, but also provides a systematic methodology for assessing confidentiality across evolving server code, especially in large, modular systems like Ejabberd.

B. Vulnerability Analysis of Confidentiality

The mathematical formulation (Figure 1 defines confidentiality in terms of the authorized access by parties (users or components) to sensitive variables. Any instance where data are accessed or transmitted by unauthorized parties represents a breach, as per our model. To validate this model in practice, we perform a static code analysis of Ejabberd's authentication modules. Specifically, we use the formal definition of the environment π, the access control list (ACL), and the access map Γ to identify violations, where sensitive values (such as passwords) are leaked outside of the authorized scope defined by Γ. We then examine whether actual function flows in the Ejabberd code correspond to these unauthorized patterns.

To investigate potential confidentiality breaches in Ejabberd, we conducted a static analysis focused on how user credentials, particularly passwords, are handled within the authentication pipeline. Our analysis was performed using RefactorErl – an open-source static source code analysis and transformation tool specifically designed for Erlang. RefactorErl enables structural and semantic querying of large Erlang codebases, making it suitable for privacy analysis in distributed messaging servers like Ejabberd.

We extended RefactorErl's `is_dirty` query to detect "side effects" in functions—defined as scenarios where data escapes the function boundary. This is especially critical in analyzing password variables, which must be confined and protected. As discussed in the following subsections and sketched in Table I, any function forwarding the password to external systems or writing it to potentially insecure storage was flagged for further inspection.

1) Password Propagation Across Modules: We analyze the propagation of the `Password` variable starting from the mod_admin_extra module, which exposes an administrative interface to set user credentials. The Password variable is first passed in the function mod_admin_extra:set_password/3:

```
set_password(User, Host, Password) ->
Fun = fun () ->
ejabberd_auth:set_password(User, Host, Password)
...end
```

Here, the password is captured by a closure and forwarded to `ejabberd_auth:set_password/3` for downstream handling. This function continues the flow by accepting and validating the password.

```
set_password(User, Server, Password) ->
case validate_credentials(User, Server, Password)
db_set_password(LUser, LServer, Password, M);
end.
```

The inner call to `db_set_password/4` represents a critical point in the flow. The password is forwarded to an external database or authentication cache. This chain of calls forms a sensitive data propagation path:
`mod_admin_extra:set_password` → `ejabberd_auth:set_password` → `db_set_password`
According to our model, passwords are sensitive variables (i.e., $(x : v) \in data(\pi)$), and any access must conform to the ACL-derived access map Γ. In `mod_admin_extra`, the `set_password/3` function passes the password through multiple layers to an external database. As these external destinations are considered unauthorised parties in our model, this constitutes a violation of Γ.

2) External Transmission via PAM (Pluggable Authentication Module): In the ejabberd_auth_pam.erl module, the function check_password/4 handles user authentication by interacting with an external system:

```
check_password(User, AuthzId, Host, Password) ->
case catch epam:authenticate(
Service, UserInfo, Password)... end.
```

This function sends the user's Password to the `epam:authenticate/3` function, which delegates authentication to an external PAM. PAM is a flexible authentication framework used in Unix-like systems. It enables applications to authenticate users by relying on pluggable modules configured at the system level, such as local password checks, LDAP, or Kerberos. This allows applications like Ejabberd to delegate authentication without implementing each mechanism internally. As PAM operates beyond the core boundaries of the application, transmitting the password to it introduces a potential security risk. This external flow constitutes a confidentiality violation under formal models such as Γ, where un-trusted components must not receive sensitive data.

3) Unprotected Communication with External Authentication Programs: In ejabberd_auth_external.erl, the function check_password_extauth/4 is implemented as follows:

```
check_password_extauth
(User, _AuthzId, Server, Password) -> ...
case extauth:check_password
(User, Server, Password) ... end.
```

Here, the password is passed to `extauth:check_password/3`, which in turn communicates with an external script or program using Erlang's `open_port` or `spawn_port` mechanism. These functions allow the Erlang runtime to interact with external operating system processes for authentication mechanisms. However, this interaction violates the application's access control policy as formalized in our security model (e.g., Γ). Specifically, the password classified as sensitive information is transmitted to an external component that is not part of the trusted network. Such a flow undermines the confidentiality guarantees of the system, as it allows sensitive credentials to leave the application boundary.

TABLE I: Summary of Confidentiality Breaches in Ejabberd

Module	Function	Operation	Breach Description
mod_admin_extra	set_password/3	External Call	Password passed to external DB/auth module
ejabberd_auth_pam	authenticate/3	External Call	Password sent in plain-text to PAM
ejabberd_auth_external	check_password_extauth/4	Spawn Port	Password exposed to external script

C. Implications

Our analysis reveals that sensitive data is often propagated across modules and sent to external systems without consistent access control enforcement. This violates confidentiality model and exposes messaging applications to privacy risks. These findings highlight the need for a systematic analysis of these threats to ensure sensitive data does not leak beyond trusted boundaries. Our static analysis ensures code level confidentiality checks but cannot capture runtime behavior or user interactions. Dynamic factors like variable states and network conditions are out of scope.

IV. Extended Privacy Analysis

In this section, we extend our analysis to detect several critical security vulnerabilities that may breach various privacy requirements, e.g., confidentiality, integrity, and anonymity, to name a few. In the following subsections, we discuss the vulnerabilities detected in the server's codebase. Table II summarizes identified Ejabberd vulnerabilities with related module, its description and risks.

A. Insecure Call

In this category, we examine vulnerabilities stemming from insecure function calls that perform unsafe de-serialization operations. These vulnerabilities typically arise when un-trusted binary data is converted into Erlang terms using functions like `binary_to_term/1`, potentially leading to unauthorized access, data tampering, or remote code execution if the inputs are not properly sanitized.

(i) **`binary to term(Data)`**: The `Data` variable stores binary-encoded tuples like {ServerHost, LocalHint} in the Ejabberd router Redis file. This introduces a security risk as improper de-serialization of this binary data can lead to vulnerabilities, such as unauthorized data access or manipulation.

(ii) **`binary to term(Pid)`**: The `Pid` contains the Process Identifier of an Erlang process. Improper handling or de-serialization could allow attackers to hijack process or gain unauthorized access to system.

(iii) **`binary to term(Val)`**: The `Val` variable holds session data in binary form. If this data is decoded improperly, it could lead to the execution of malicious code. The insecure calls like `binary_to_term/1` operations from Ejabberd is shown in Figure. 3.

B. Insecure File Operations

Insecure file operations involve accessing/executing files without proper validation or access control, leading to expose sensitive data or the execution of arbitrary code.

```
decode_session_list(Vals) ->
  [binary_to_term(Val) || {_, Val} <- Vals].

fun({Pid, Data}) ->
    {ServerHost, LocalHint} = binary_to_term(Data),
    #route{domain = Domain,
      #route{domain = Domain,
         pid = binary_to_term(Pid),
         server_host = ServerHost,
```

Fig. 3: Insecure Calls

(i) **`file:consult(File)`**: The roster file stores sensitive user information such as contact details, presence state, and group memberships. If improperly accessed, this file could expose sensitive data, leading to privacy violations. Figure 4 shows insecure access to sensitive user data from the roster file.

```
push_roster_all(File) ->
    {ok, [Roster]} = file:consult(File),
    subscribe_all(Roster).
```

Fig. 4: Insecure File Operation

C. Insecure OS Call

Insecure OS calls occur when an application runs operating system commands without properly checking or cleaning the input.

(i) **OS Injection**: The system command functions like `os:cmd/1` are vulnerable to OS command injection. User input is passed directly into system commands without proper sanitization, enabling an attacker to inject malicious commands. An example of such an insecure OS call in Ejabberd is illustrated in Figure. 5.

(ii) **Denial of Service (DoS)**: If the `Cmd` or `LibDir` variables are maliciously crafted, they can be exploited to cause system overload, file deletion, or system crash.

```
compile_c_files(LibDir) ->
    case file:read_file_info(filename:join(LibDir, "c_src/Makefile")) of
        {ok, _} ->
            os:cmd("cd "++LibDir++"; make -C c_src");
        {error, _} ->
            ok
    end.
file:set_cwd(SrcDir),
filelib:ensure_dir(filename:join("deps", ".")),
lists:foreach(fun({App, Cmd}) ->
            io:format("Fetching dependency ~s: ", [App]),
            Result = os:cmd("cd deps; "++Cmd++"; cd .."),
            io:format("~s", [Result])
    end, Deps),
file:set_cwd(CurDir)
```

Fig. 5: Insecure OS Call

TABLE II: Summary of Extended (Privacy) Vulnerabilities Analysis

Vulnerability	**Identified Code\Module**	**Description**	**Issues**
		Vulnerable Variable and its Value	
Unsafe external executables	open_port(spawn, Path, [packet, 2])	**Path** refers to a program or command that runs to communicate to external OS process: which is extauth program: **python3 extauth.py** to handle authentication requests	If Path is un-trusted or user-controlled: Arbitrary Code Execution, Command injection [12]
Insecure file operation	file:script(Script) file:consult(File)	**Script** variable is the path to the configuration script **rebar.config.script** **File** refers to the data of roster table used by Ejabberd	Sensitive Data Exposure Arbitrary code execution [13]
Unsafe network	net_kernel:connect_node(Node)	**Node** represents name with domain address	Unauthorized access to distributed node and host machine, can access sensitive data
Insecure calls	binary_to_term(Pid) binary_to_term(Data) binary_to_term(Val) binary_to_term(<< $131, 103, 100(size(NodeBin))$: $16, NodeBin/binary, 0:72 >>$))	**Pid** is an identifier of an Erlang process **Data** is a binary decoded tuple {ServerHost, LocalHint}, used to create a route record **Val** is the value part of each key-value pair, while the exact value s stored in Redis, the key represents either USKey, SIDKey, ServKey, USSIDKey **NodeBin** represents binary format of node name e.g., "nodename@localhost".	Remote code execution
Insecure atom creation	list_to_atom(ElixirModule) list_to_atom(S) list_to_atom(NodeString) list_to_atom(Scheme) list_to_atom(STable) list_to_atom(FromString) list_to_atom(atom_to_list(Mod) ++ "_cache") list_to_atom(NameString)	**NameString** refers to name of environment variable starting with EJJABERD_MACRO **S** refers to server name **Scheme** represents the protocol e.g., mqtt, ws, https	Arbitary code execution, Crash leading to DOS, subsequent write operations can produce undefined or unexpected results
Insecure OS call	os:cmd("cd deps; "++Cmd++"; cd ..") os:cmd("cd "++LibDir++"; make -C c_src");	**Cmd** contains actual command to be executed in deps directory **LibDir** contains a path to base directory and package/module name	Command Injection [14] [15], Denial of service, Arbitrary code execution if improper input validation [13]
Insecure ETS traversal / Insecure concurrency	ETS:first(caps_features_tmp)) ETS:next(caps_features_tmp, NodePair)) ETS:first(privacy_list_tmp)) ETS:next(privacy_list_tmp, US)) ETS:next(Component, Host, << "" >>) ETS:next(?MODULE, Key))	**Privacy_list** contains rules defined by $< item/ >$ elements. Each rule specifies the type (JID, group, etc.), the target value, the action (allow or deny), order of processing [16]. **US** = {LUser, LServer}	Race Condition [17], Data corruption, Denial of service
Decommissioned crypto	crypto:sign(eddsa, none, Msg, [PrivKey, ed25519]) crypto:verify(eddsa, none, Msg, Signature, [VerifyKey, ed25519])	To find obsolete functions in the crypto module, it's sufficient to identify the function call's location; checking the parameters is not necessary. [13]	Misuse of cryptographic operations, Improper key management
Insecure compile operations	code:load_binary(rebar_config, "mock", Bin) compile:file(File, Options) code:load_binary(Mod, "nofile", Code) code:load_file(Module) compile:file(File, Options)	**rebar_config** represents configuration file containing different dependencies of external libraries	Improper validation of input or file paths

D. Insecure ETS Traversal

It refers to inconsistent read or write operations on Erlang Term Storage (ETS) tables, where invalid or unauthorized access can result in data integrity breach, race conditions, or potential data tamper, leading to security vulnerabilities.

- **Race Condition**: The race condition vulnerability arises when the `ETS:next/2`, `ETS:first/2` functions are used to fetch data from ETS tables without proper synchronization as shown in Figure 6. If concurrent operations insert or delete entries during traversal, it can cause crashes or result in inconsistent data.

```
import_next(LServer, DBType, NodePair) ->
    Features = [F || {_, F} <- ets:lookup(caps_features_tmp, NodePair)],
    Mod = gen_mod:db_mod(DBType, ?MODULE),
    Mod:import(LServer, NodePair, Features),
    import_next(LServer, DBType, ets:next(caps_features_tmp, NodePair)).

import_stop(_LServer, DBType) ->
    import_next(DBType, ets:first(privacy_list_tmp)),
    ets:delete(privacy_default_list_tmp),
    ets:delete(privacy_list_data_tmp),
    ets:delete(privacy_list_tmp),
    ok.
```

Fig. 6: Insecure ETS Traversal

E. Unsafe External Executables

These occur when Erlang invokes external programs (e.g., via `open_port/2` or `os:cmd/1`) without proper input validation leading to following risks:

(i) **Arbitrary Code Execution**: In `ejabberd_auth_external`, the function `spawn_port/1` is used to launch an external python script `extauth.py` for authentication purposes. This poses a risk when inputs are not validated, as un-trusted data passed to external executables can lead to arbitrary code execution, whose example from Ejabberd is shown in Figure 7.

(ii) **Lack of Input Validation**: The `extauth.py` script reads input directly from `sys.stdin`, making it vulnerable to unauthorized access and improper validation allowing attackers to bypass authentication.

```
-spec start_port(string()) -> {port(), integer() | undefined}.
start_port(Path) ->
    Port = open_port({spawn, Path}, [{packet, 2}]),
    link(Port),
    case erlang:port_info(Port, os_pid) of
    {os_pid, OSPid} ->
        {Port, OSPid};
```

Fig. 7: Unsafe External Executable

F. Decommissioned Crypto Functions

A decommissioned cryptographic function in Erlang refers to an encryption or hashing algorithm that has been marked as obsolete or insecure due to identified vulnerabilities.

(i) **Privacy Threat**: The use of deprecated cryptographic primitives from OpenSSL, such as `block encrypt/3/4` and `cmac/3/4`, exposes the system to potential weaknesses. These low-level functions require developers to manage cryptographic keys manually, increasing the risk of key compromise if not properly protected or rotated. The use of decommissioned crypto functions from Ejabberd is shown in Figure 8.

```
    KeyID = <<"ed25519:", KeyName/binary>>,
    Sig = crypto:sign(eddsa, none, Msg, [PrivKey, ed25519]),
    Sig64 = base64_encode(Sig),
Msg = mod_matrix_gw:encode_canonical_json(JSON2),
crypto:verify(eddsa, none, Msg, Signature, [VerifyKey, ed25519])
```

Fig. 8: Decommissioned Crypto

G. Insecure Atom Creation

Insecure atom creation refers to the unsafe generation of atoms from un-trusted input using functions like `list_to_atom/1` or `binary_to_atom/2`.

(i) **Atom Table Overflow**: The environment variable parsing function `parse/1` creates atoms based on input, which are stored in the global atom table. Since atoms are not garbage collected in Erlang, if the table fills up, the system may crash, resulting in a denial of service (DoS). The atom exhaustion risks and its prevention methods are discussed in [18]. Figure.9 shows such examples from Ejabberd codebase.

```
   Attrs = mnesia:table_info(Name, attributes),
   TmpTab = list_to_atom(atom_to_list(Name) ++ "_backup"),
Proc = list_to_atom(atom_to_list(Module) ++ "_sup"),
ChildSpec = {Proc, {ejabberd_tmp_sup, start_link, [Proc, Module]},
 C = ejabberd_commands:get_command_definition(
                    list_to_atom(NameString), Version),
 #ejabberd_commands{name = Name,
```

Fig. 9: Insecure Atom Creation

H. Insecure Compile Operations

Insecure compilation refers to runtime code using untrusted or dynamically provided code, often through functions like `compile:file/1` or `code:load binary/3`.

(i) **Code Loading from Binary**: Using the `code:load binary/3` function to load compiled modules at runtime presents a significant security risk as shown in Figure. 10. Without verifying the integrity of the binary, attackers can substitute a malicious version, potentially compromising the system.

```
     Options = [{outdir, Dest} | ErlOptions],
     case compile:file(File, Options) of
         {ok, Module} -> {ok, Module};
{module, Mod} = code:load_binary(Mod, "nofile", Code),
ok
            code:delete(Module),
            code:load_file(Module),
```

Fig. 10: Insecure Compile Operations

V. Related Work

Privacy analysis methods are primarily categorized into static and dynamic analysis. Static analysis tools such as FlowDroid [4] use static taint analysis to detect sensitive data leaks early in development. FlowDroid is highly effective for lifecycle aware, object-sensitive taint analysis, but it is not designed for backend privacy verification or for

formally modeling access control policies over authentication modules. Compared to Flowroid, Our approach extends static analysis beyond data flows by integrating formal access control modeling for confidentiality. Dynamic analysis tools like DroidMiner [19] and Mobile Privacy Compliance [20] can detect behaviors that static analysis might overlook but require test harnesses, instrumentation, and execution environments. In contrast, our work intentionally avoids dynamic execution requirements. While dynamic tools provide valuable runtime insight, they often lack the scalability and early phase detection advantages that static analysis offers in large server-side codebases. Model driven privacy analysis relies on formal models to represent a system's structure and behavior. For instance, the approach by Pedroza et al. [21] introduces privacy by design using formal verification, typically at the system architecture level. Although our approach differs in the way that we apply formal reasoning directly at the implementation level rather than during high level design. Our access control model is embedded into the source code analysis workflow, enabling practical vulnerability detection grounded in semantic program structure. In contrast, data driven approaches analyze system logs or network traces to detect privacy leaks. Tools like PriME [22] combine machine learning with trace analysis to measure privacy risks dynamically. However, such approaches depend heavily on data quality and are not easily generalizable across systems with different architectures or data collection capabilities. Our static, code-centric method avoids external dependencies and applies across systems like Ejabberd. To our knowledge, it is the first to unify confidentiality, anonymity, and consent detection within a single formal model at the server implementation level.

VI. Conclusion

This study examines privacy in the Ejabberd server, specifically focusing on confidentiality breaches. Using our unified privacy analyzer that is developed on top of the RefactorErl tool, we have identified several privacy vulnerabilities, including the exposure of passwords to external services and unsafe storage practices. These findings underscore the need for stronger privacy controls in open-source communication platforms like Ejabberd. While focused on confidentiality, our extensible model supports broader privacy goals such as anonymity and consent within a unified evaluation framework. Future work focuses on extending these tests to other privacy requirements, helping developers build privacy aware communication systems.

References

[1] "Impact Report 2022 - Internet Society — internetsociety.org." https://www.internetsociety.org/impact-report/2022/. [Accessed 10-04-2025].

[2] "Cost of a data breach 2024 — IBM — ibm.com." https://www.ibm.com/reports/data-breach. [Accessed 10-04-2025].

[3] B. Lokhande and S. Dhavale, "Overview of information flow tracking techniques based on taint analysis for android," in *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 749–753, IEEE, 2014.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[5] B. Blanchet *et al.*, "An efficient cryptographic protocol verifier based on prolog rules. 2014," *doi*, vol. 10, pp. 82–96.

[6] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pp. 1–6, 2017.

[7] K. Wuyts, L. Sion, and W. Joosen, "Linddun go: A lightweight approach to privacy threat modeling," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, pp. 302–309, 2020.

[8] A. Hajdu, M. Marescotti, T. Suzanne, K. Mao, R. Grigore, P. Gustafsson, and D. Distefano, "Inferl: scalable and extensible erlang static analysis," in *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*, pp. 33–39, 2022.

[9] E. L. University, "RefactorErl - Refactoring Erlang Programs — plc.inf.elte.hu." https://plc.inf.elte.hu/erlang/. [Accessed 08-04-2025].

[10] B. Baranyai, I. Bozó, and M. Tóth, "Supporting secure coding with refactorerl," *Submitted to the ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, 2020.

[11] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-113.pdf. [Accessed 08-10-2024].

[12] "Spawning external executables — erlef.github.io." https://erlef.github.io/security-wg/secure_coding_and_deployment_hardening/external_executables. [Accessed 06-03-2025].

[13] B. Baranyai, I. Bozó, and M. Tóth, "Supporting secure coding with refactorerl," *Submitted to the ANNALES Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica*, 2020.

[14] "A03 Injection - OWASP Top 10:2021 — owasp.org." https://owasp.org/Top10/A03_2021-Injection/. [Accessed 03-03-2025].

[15] "CWE - CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (4.16) — cwe.mitre.org." https://cwe.mitre.org/data/definitions/78.html#Vulnerability_Mapping_Notes_78. [Accessed 03-03-2025].

[16] "XEP-0016: Privacy Lists — xmpp.org." https://xmpp.org/extensions/xep-0016.html#protocol-syntax. [Accessed 04-03-2025].

[17] "CWE - CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') (4.16) — cwe.mitre.org." https://cwe.mitre.org/data/definitions/362.html. [Accessed 04-03-2025].

[18] "Preventing atom exhaustion — erlef.github.io." https://erlef.github.io/security-wg/secure_coding_and_deployment_hardening/atom_exhaustion. [Accessed 10-03-2025].

[19] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Computer Security-ESORICS 2014. Proceedings, Part I*, pp. 163–182, Springer, 2014.

[20] J. Lei, Y. Wu, N. Hu, J. Tao, Y. Wang, H. Wang, and M. Fan, "Dynamic testing for mobile privacy compliance," in *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 113–114, 2024.

[21] G. Pedroza, V. Muntes-Mulero, Y. S. Martin, and G. Mockly, "A model-based approach to realize privacy and data protection by design," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 332–339, IEEE, 2021.

[22] A. Hassanpour and B. Yang, "Prime: A novel privacy measuring framework for online social networks," in *2022 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 518–525, IEEE, 2022.