WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

Stefano Calzavara and Riccardo Focardi, Università Ca' Foscari Venezia;
Matteo Maffei and Clara Schneidewind, TU Wien;
Marco Squarcina and Mauro Tempesta, Università Ca' Foscari Venezia

Usenix Security 2018

Minkyung Park

mkpark@mmlab.snu.ac.kr

May 6, 2021

Contents

- Motivation
- OAuth 2.0 protocol
- WPSE
- Evaluation
- Conclusion

Web protocols

- Web protocols are usually to implement authentication and authorization at remote servers
  - e.g., OAuth 2.0, OpenID Connect, SAML 2.0, etc.

- Designing and implementing web protocols is a error-prone task
  - Many vulnerabilities are reported

- Why? web browser is **agnostic** to the web protocol **semantics**
  - Start/end of the protocol
  - The order in which messages should be processed
  - The confidentiality/integrity guarantees desired for a protocol run

Countermeasure?

- Major service providers try to aid developers to correctly integrate web protocols
  - Provide JavaScript APIs
- Web developers are ***not forced*** to use them
  - Still can use them incorrectly
- The APIs ***still fail***
  - The APIs themselves do not implement the best security practices

Web Protocol Security Enforcer (WPSE)

- To strengthen the security guarantees of web protocols, browsers are **extended with a security monitor**
- The security monitor **enforces the compliance** of browser behaviors
- The browser is aware of the intended protocol flow by given **protocol specifications**
  - Protocol specification can be written and verified once, uniformly enforced at a number of different websites
- Web applications are automatically protected against bugs or vulnerabilities on the browser-side

Background on OAuth 2.0

- Resource owners can grant third-parties **controlled access to resources** at remote servers
  - It is also used for authenticating the resource owner by giving third parties to the owner's identity (Single Sign-On)
  - Third-party applications are referred as relying party (RP)
  - Remote servers storing the resources are referred as identity provider (IdP)

- The OAuth 2.0 specification defines four grant types (modes)
  - **Authorization code**, implicit, resource owner password credentials, and client credentials

Authorization code mode

- RP submits **authorization code** to prove its access grant
  - In implicit mode, instead of granting an authorization code to RP, the IdP provides an access token directly

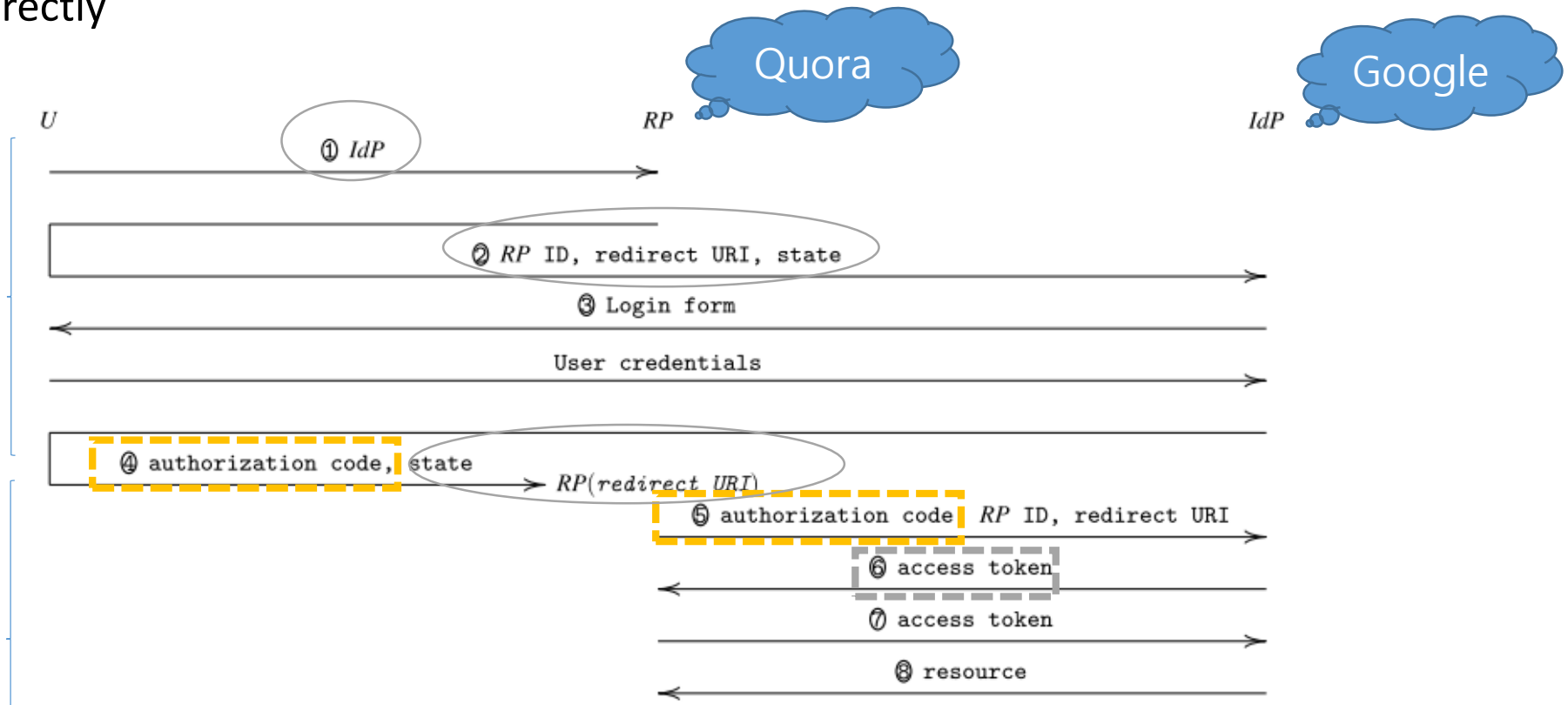
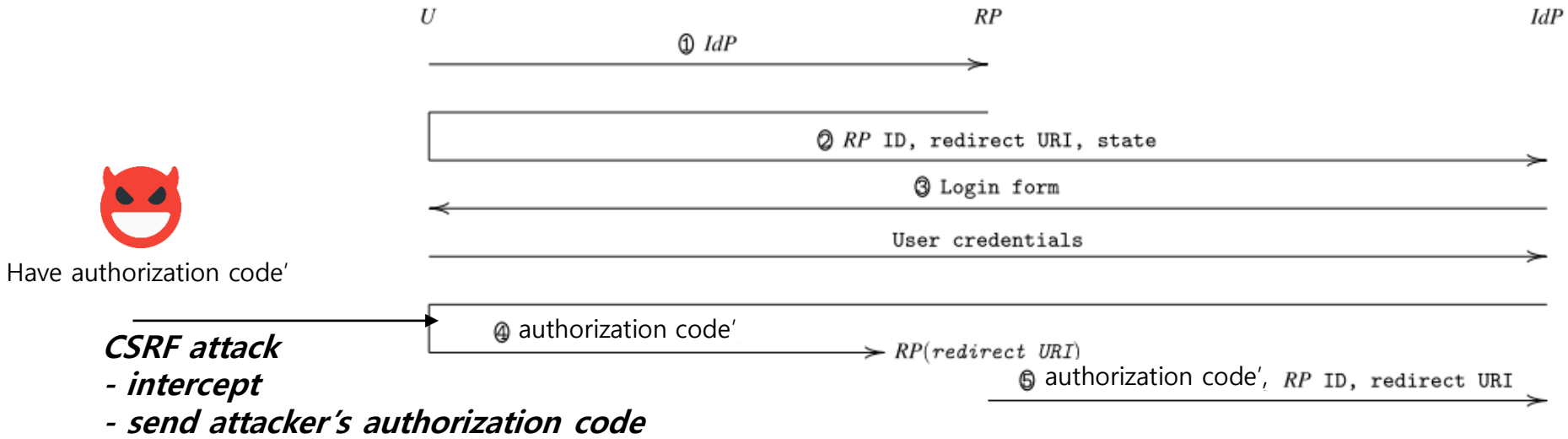


Figure 1: OAuth 2.0 (authorization code mode).

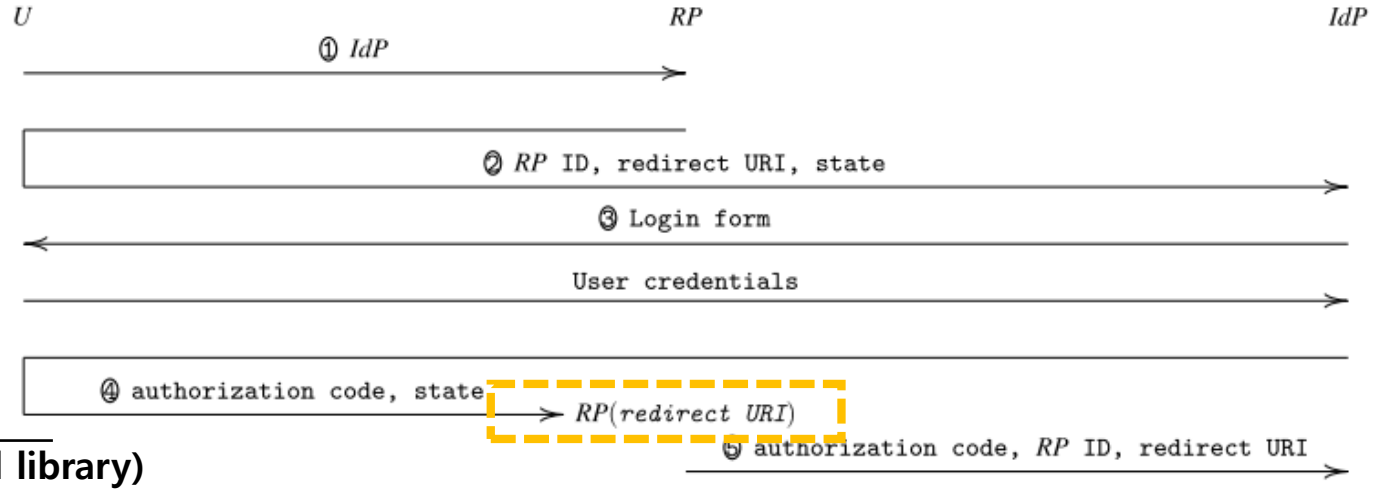
Challenges #1: protocol flow

- Protocols are specified in terms of a number of sequential message exchanges
  - The browser is not forced to comply with the intended protocol flow
- Attacks to skip messages or to accept them in a wrong order are possible
- Example: session swapping attack
  - Completing a social login in the user's browser that was not initiated before
  - When RP does not provide the state parameter at step (2), it is possible to force the honest user's browser to authenticate as the attacker
    - State parameter: a value bound to a session (e.g., session hash)



Challenges #2: secrecy of messages

- The security of protocols relies on the confidentiality of cryptographic keys and credentials
  - However, the browser is not aware of which data must be kept secret
- Example: state leak attack
  - If the page loaded at the redirect URI loads a resource from a malicious server, the state parameter and the authorization code can be leaked in the Referer header

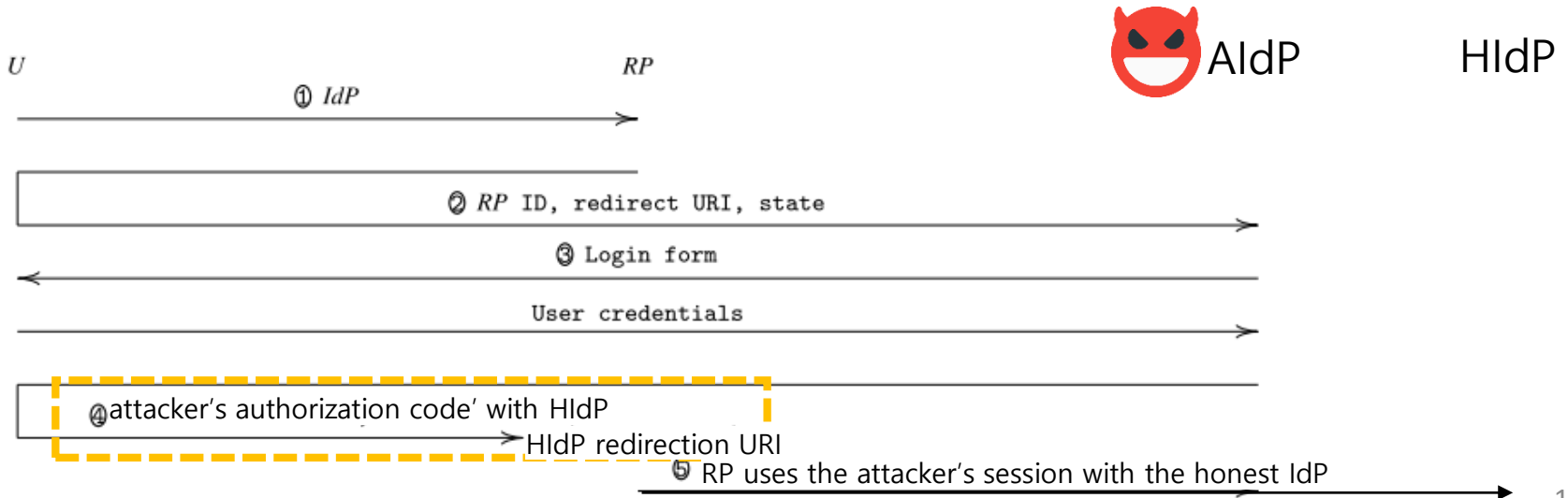


3rd party

Send req. (e.g., load ad library)
(Referer header: containing authorization code, state)

Challenges #3: integrity of messages

- The integrity of the messages they send should be ensured
  - However, the browser cannot perform these checks
- Example: Naïve RP session integrity attack
  - RP supports distinguish a selected IdP using different redirect URIs
  - An attacker controlling a malicious IdP (AIdP) can confuse the RP about which IdP is being used and force the user's browser to login as the attacker

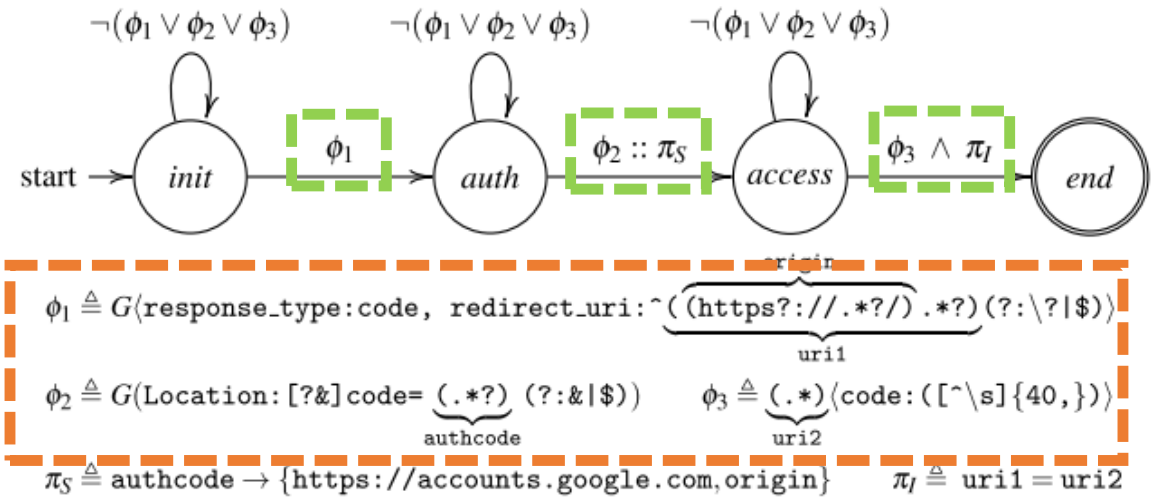


Design overview

- The Web Protocol Security Enforcer (WPSE) is the browser-side security monitor
  - The prototype is implemented as a browser extension
- Web protocols are given in XML to WPSE
  - Web protocols are described in terms of ***the HTTP(S) exchanges*** observed by the web browser
- WPSE checks the given protocol specifications over messages in runtime
  - If any violation is detected, the corresponding message is not processed and the protocol run is aborted

Protocol specification

- It defines the **syntactic structure,** the expected **order of the messages**, and the required **secrecy and integrity policies**
  - Syntactic structure is described using regular expressions
- It can be represented in **finite state automata**
  - Each state: one stage of the protocol execution
  - State transition: sending HTTP(S) requests and receiving HTTP(S) responses



e<a>: HTTP request
e(h): HTTP response
- e: remote endpoint
- a: a list of parameters
- h: a list of headers

Figure 2: Automaton for OAuth 2.0 (authorization code mode) where G is the OAuth endpoint at Google.

Protocol specification

- Secrecy policy: which parts of the HTTP(S) responses must be confidential
  - $x \rightarrow S$: the value x can be disclosed only to the origins specified in the set S
  - e.g., π_S: “authcode” is disclosed only to “origin” and “https://accounts.google.com”
- Integrity policy: incoming messages is compared with the messages in previous steps
  - e.g., π_I: uri1 should be same to the uri2

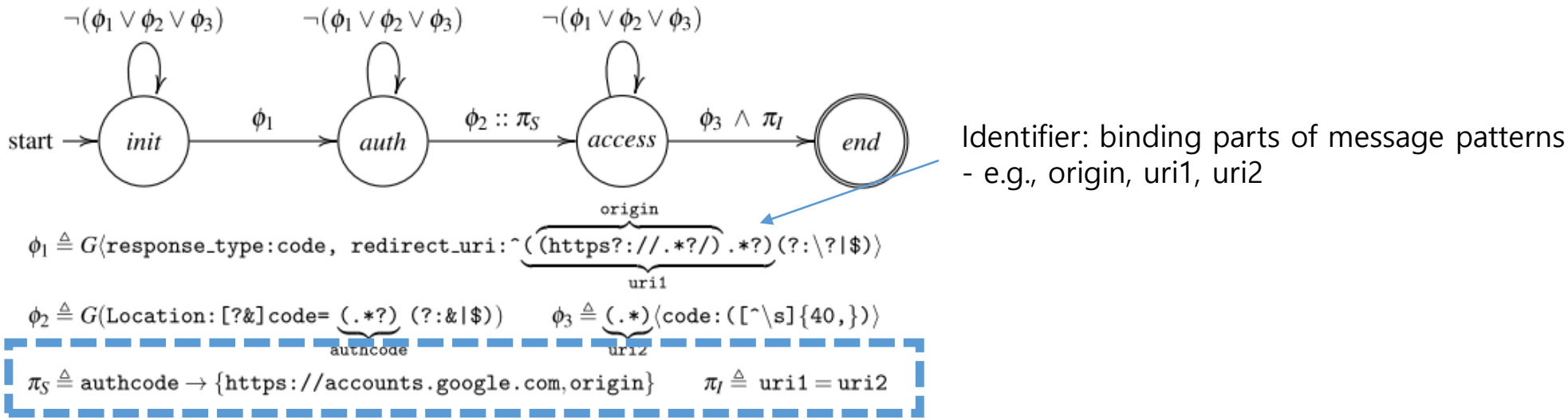


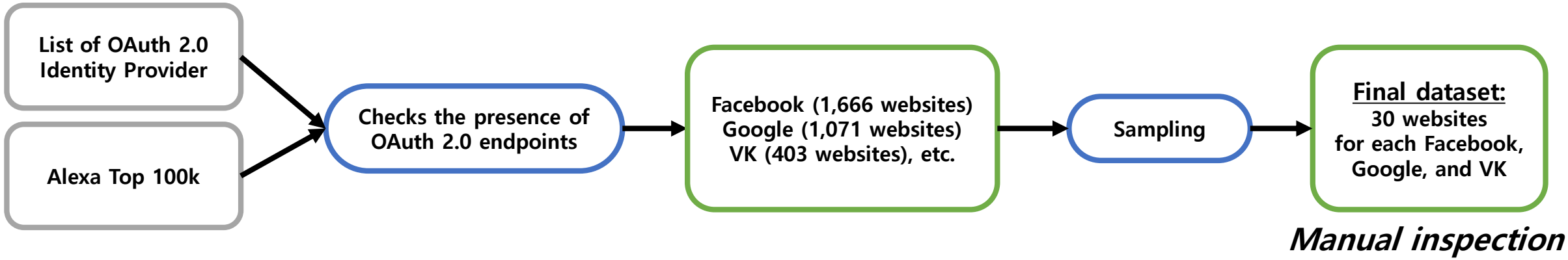
Figure 2: Automaton for OAuth 2.0 (authorization code mode) where G is the OAuth endpoint at Google.

Identifier: binding parts of message patterns - e.g., origin, uri1, uri2

Experimental setup

- To assess the **security benefit** and the **compatibility** of WPSE, OAuth 2.0 protocol is evaluated on existing websites in the wild

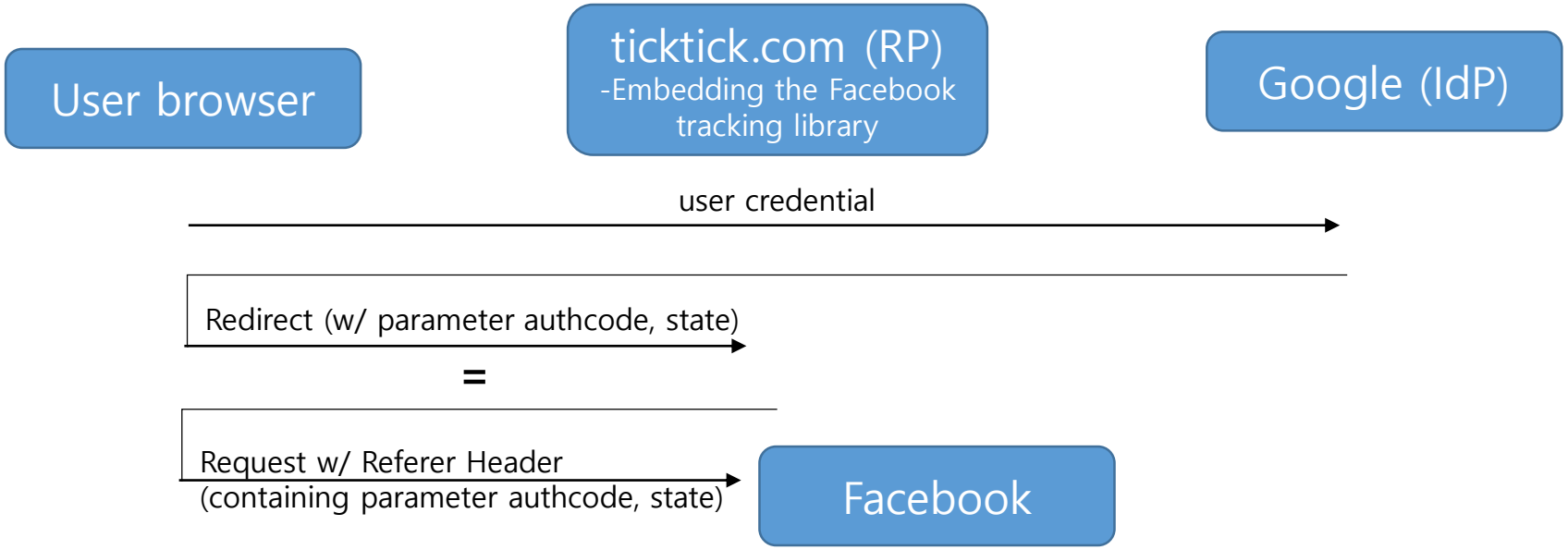
- Crawler: automatically identify existing OAuth 2.0 implementations



- Specification: the most common use case is modeled, a protocol specification is devised for each identity provider
  - There is slight differences between identity providers in practice
  - e.g., the use of the response_type parameter is mandatory at Google, but not at Facebook and VK

Security analysis

- The extension prevented the leakage of sensitive data on 4 different relying parties
- In all cases, the violations are due to the presence of tracking or advertisements libraries
  - Such as Facebook Pixel, Google AdSense, Heap, and others



Security analysis

- 55 out of 90 websites have been found affected by the lack or misuse of the state parameter
- 41 websites do not support it while 14 websites miss the security benefit
  - Using a predictable or constant string as a value
- The situation is caused by the IdPs not setting the parameters as mandatory
  - State parameter is recommended by Google and VK, and mandatory by Facebook (according to their documentation)

Compatibility analysis

- The usage of WPSE did not impact in a perceivable way the browser performance or the time to load webpages

- 81 websites were navigated flawlessly, but in 9 websites, the protocol run were not able to be successfully completed
  - 2 websites were related to the use of the Gigya social login provider
    → Fixed by writing an XML specification for the Gigya
  - 7 websites were because of the deviation from the OAuth 2.0 specification
    → Not fixed; may introduce security flaws

Conclusion

- WPSE is the first browser-side security monitor to address the security challenges of web protocols
- Given protocol specifications, WPSE enforces the web browser to follow those security requirements at runtime
  - i.e., protocol flows, secrecy, and integrity
- Existing OAuth 2.0 implementations show that the security benefits and compatibility of WPSE