IPSJ Transactions on Advanced Computing Systems Vol.6 No.1 45–55 (Jan. 2013)

Regular Paper

Automatically Checking for Session Management Vulnerabilities in Web Applications

Yusuke Takamatsu[1,a)] Yuji Kosuga[2] Kenji Kono[1,3]

Received: July 4, 2012, Accepted: October 17, 2012

Abstract: Many web applications employ session management to keep track of visitors' activities across pages and over periods of time. A session is a period of time linked to a visitor, which is initiated when he/she arrives at a web application and it ends when his/her browser is closed or after a certain time of inactivity. Attackers can hijack a user's session by exploiting session management vulnerabilities by means of session fixation and cross-site request forgery attacks. Even though such session management vulnerabilities can be eliminated in the development phase of web applications, the test operator is required to have detailed knowledge of the attacks and to set up a test environment each time he/she attempts to detect vulnerabilities. We propose a technique that automatically detects session management vulnerabilities in web applications by simulating real attacks. Our technique requires the test operator to enter only a few pieces of basic information about the web application, without requiring a test environment to be set up or detailed knowledge of the web application. Our experiments demonstrated that our technique could detect vulnerabilities in a web application we built and in seven web applications deployed in the real world.

Keywords: web application security, session management, vulnerability, session fixation, cross site request forgery

1. Introduction

Most recent web applications have employed session management to keep track of visitors' activities over inherently stateless protocols such as HTTP. A session identifier (SID) is issued to a visitor on his first visit to a web application to keep track of his activities. The session ID is an attractive target for malicious attackers because they can masquerade as visitors if they visit a web application with a visitor's session ID.

Session fixation [1] and Cross-site Request Forgery (CSRF) [2] are major attacks inflicted on the session management mechanisms. An attacker in a session fixation attack prepares a session ID with which a victim is forced to log into a target web application. After a victim has logged in, an attacker can access the web application with the victim's privileges. An attacker in a CSRF (also known as a one-click attack, session-riding, or XSRF) attack forces a victim to execute unwanted actions on the web application for which the victim is currently authenticated. A victim with a little help from social engineering, is forced to follow a link or execute a script that executes an action chosen by the attacker, which is successfully processed on the web application since the victim is currently authenticated.

Session fixation and CSRF are not difficult to prevent in the development phase of web applications, because the countermeasures against them are well-known and not hard to incorporate into web applications. However, a large number of vulnerabilities have been reported. According to a report from WhiteHat Security [3], 14% of web applications were vulnerable to session fixation and 24% were exposed to CSRF in the first quarter of 2011. These vulnerabilities are real threats. A digital currency, Bitcoin [4], declined sharply in 2011 due to a CSRF vulnerability in a currency exchange application called Mt.Gox [5]. The vulnerability in Mt.Gox caused the victims to exchange the Bitcoin currency in Mt.Gox.

There are two reasons that a large number of session management vulnerabilities are not eliminated during the development of web applications. First, web developers are occasionally unskillful and not familiar with session fixation or CSRF attacks. Eliminating all vulnerabilities in web applications is practically impossible because the test phase cannot be comprehensive or thorough enough due to severe time-to-market constraints. Web developers are sometimes not familiar with all kinds of web vulnerabilities. Catching up with state-of-the-art web attacks and countermeasures to these is difficult due to the increased complexity of attack methods as well as web applications. Second, it is a tedious, time-consuming, and daunting task to check for potential vulnerabilities in web applications, and web developers must set up a test environment and prepare malicious scripts to emulate attacks and so on.

We propose a technique in this paper of automatically checking web applications for session management vulnerabilities, especially session fixation and CSRF attacks. We implemented our technique as a plug-in of Amberate [6], [7], which is an extensible framework for checking web applications for various security holes. Amberate automatically checks for the presence of nu-

[1] Keio University, Yokohama, Kanagawa 223–8522, Japan
[2] Everforth Co., Ltd., Meguro, Tokyo 152–0035, Japan
[3] JST CREST
[a)] yusuke@sslab.ics.keio.ac.jp

A preliminary version of this paper will appear in IEEE 10th Annual International Conference on Privacy, Security and Trust.

merous kinds of web vulnerabilities such as SQL injection and Cross-Site Scripting (XSS). It is designed to be used in the development phase of web applications and thus, assumes internal knowledge of web applications is available because the developers can provide these pieces of information. Amberate provides a set of APIs with which a plug-in can be implemented that checks for a particular vulnerability in web applications. Amberate currently provides the plug-ins for SQL injection, XSS, JavaScript Hijacking, and directory traversal.

The primary contributions of this paper are twofold. First, it proposes a new technique for detecting session fixation and CSRF vulnerabilities. Our approach carries out pseudo attacks to detect these vulnerabilities. Discussed later in the paper, this approach improves the detection accuracy of these vulnerabilities. Second, our technique is automated by being incorporated into Amberate. Automated checking by Amberate brings about several advantages. Even if test operators are not familiar with a specific kind of vulnerability, Amberate automatically checks all kinds of vulnerabilities (if the plug-in for checking that vulnerability has been provided). It also releases test operators from the tedious and daunting task of having to set up test environments for various kinds of vulnerabilities.

We applied Amberate to seven real web applications such as *Mambo* [8] and *Joomla* [9] to demonstrate its effectiveness. Amberate detected six session fixation vulnerabilities and 11 CSRF vulnerabilities. These applications are not trivial as they consisted of 34,439 to 233,978 lines of code.

The remainder of this paper is organized as follows. We describe the background, session management vulnerabilities in Section 2. Section 3 explains our system and Section 4 presents our experimental results. Section 5 has a review and discussion of related work. Finally, we conclude the paper in Section 6.

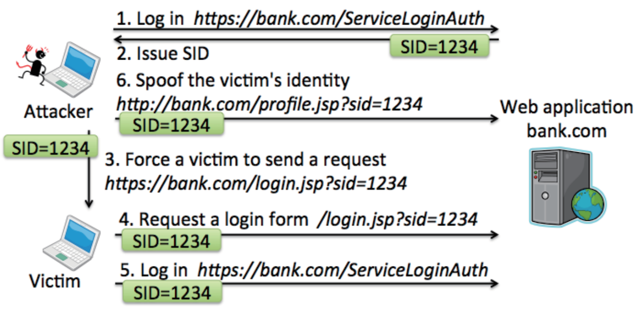
2. Session Management Vulnerabilities

We present session management and session-related vulnerabilities that we focused on in this section. Session fixation and CSRF are briefly explained, in particular, and the countermeasures against them are described.

2.1 Session Management in Web Applications

Many recent web applications have employed session management to keep track of visitors' activities over inherently stateless protocols such as HTTP. A session is associated with a user on a web application, and the session information is managed on the server-side. This information, e.g., contains a user's login status, the URI of the last visited page, and a shopping history. The web application identifies users with the session identifier (SID) that is embedded in the request.

An SID is assigned on the first visit on a web application. It is embedded in a URI, an HTTP parameter, or a cookie and is returned to the visitor as part of the response message. The received SID is sent back to the web application when the site is visited again. If an SID is embedded in a URI, it is sent back to the web application when the URI is clicked. If it is embedded in an HTTP parameter, it is sent back when the form is submitted. If it is embedded in a cookie, the SID is sent back as part of the HTTP headers.



1. Log in https://bank.com/ServiceLoginAuth
2. Issue SID
SID=1234
6. Spoof the victim's identity
http://bank.com/profile.jsp?sid=1234
SID=1234
Attacker
SID=1234
3. Force a victim to send a request
https://bank.com/login.jsp?sid=1234
Web application
bank.com
4. Request a login form /login.jsp?sid=1234
SID=1234
Victim
5. Log in https://bank.com/ServiceLoginAuth
SID=1234

Fig. 1 Session fixation.

A vulnerability in session management is an attractive target for attackers because they can spoof a victim's identity and conduct a variety of actions on the vulnerable web application.

2.2 Session Fixation

Session fixation forces a visitor to use an SID that an attacker has prepared. A problem with session fixation is that an SID that a web application has issued to a visitor can be used by other visitors. Since web applications identify visitors by their SIDs, an attacker can masquerade as a visitor after the visitor has logged into the web application with the attacker's SID. This enables the attacker to take control over the victim's account. By masquerading as the visitor, the attacker can obtain a victim's personal information, and shop as the victim.

Figure 1 shows an example of session fixation attacks. An attacker logs into a web application (Steps 1 & 2) to obtain an SID. Note that the attacker is required to log out of the web application to prevent the victim from logging into the attacker's session afterward. The attacker uses some kind of social engineering to force the victim to use the SID. For example, the attacker embeds the SID into a hyperlink to lure the victim into clicking onto it (Step 3). The victim then sends a request containing the SID (Step 4), and logs into the web application (Step 5). The attacker sends a request to the web application with the SID (Step 6), and the web application recognizes the request to be a victim's request.

Session fixation is caused because a web application continues to use the SID issued for the previous login after a user has logged in again. In the example in Fig. 1, the SID is not changed after the victim has logged in. Therefore, requests from the attacker are recognized as those from the victim. A new SID must be assigned each time a visitor logs into a web application to prevent session fixation. If a new SID is issued for each login, the SID issued to an attacker in Step 1 differs from that issued to a victim. As a result, session fixation becomes impossible.

2.3 Cross-Site Request Forgery (CSRF)

An attacker in CSRF forces a victim to execute any actions on behalf of the attacker at the target web application. If an attacker can have a victim's browser send a request on behalf of the attacker while the victim is logging in a web application, the attacker can perform the request with the privileges of the victim. WhiteHat security [3] reports that every web application has the potential to be vulnerable to CSRF.

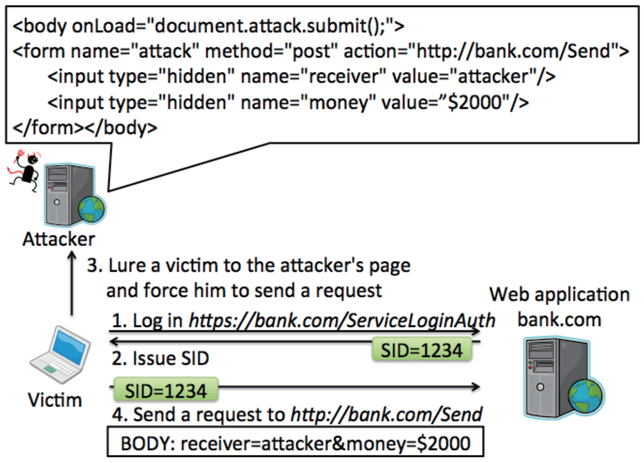

```
<body onLoad="document.attack.submit();">
<form name="attack" method="post" action="http://bank.com/Send">
    <input type="hidden" name="receiver" value="attacker"/>
    <input type="hidden" name="money" value="$2000"/>
</form></body>
```
Attacker
3. Lure a victim to the attacker's page and force him to send a request
Web application bank.com
1. Log in https://bank.com/ServiceLoginAuth
2. Issue SID
SID=1234
Victim
SID=1234
4. Send a request to http://bank.com/Send
BODY: receiver=attacker&money=$2000

Fig. 2 CSRF.

Figure 2 shows an example of CSRF attacks. First, a victim logs into a web application and receives an SID (Steps 1 & 2). The attacker lures the victim to force him/her to issue a request that the attacker prepared to the attacker's page containing a malicious script that executes some actions at the web application (Step 3). The attacker in this example prepares a script that requests a money transfer to the attacker. The malicious script makes the victim's browser send a request and the web application executes the requested action (Step 4). When this script is executed on the victim's browser, the request for a money transfer is automatically sent to bank.com with the victim's SID.

CSRF can be avoided if a web application distinguishes requests that the visitor intended to send and those that he did not. There are three well-known techniques for doing so: 1) using a secret token, 2) embedding the SID, and 3) checking the referer header.

A secret token is a unique string embedded into an HTML page. It is sent to a web application to identify where a request came from. A web application only handles a request when it has a valid secret token. Since an attacker cannot forge a secret token, the requests made by the attacker cannot be accepted by the web application.

The technique of embedding an SID into the URL or a hidden field in an HTML page also effectively prevents CSRF. Web applications can distinguish requests by checking if requests have a victim's SID or not. Requests that the visitor intended are embedded with the victim's SIDs. Web applications only handle requests that contain valid SIDs. Requests that the attacker generated are not embedded with the victim's SIDs since an attacker cannot generate with the victim's SID. Any request made by the attacker's script is not handled by the web application. This countermeasure cannot prevent CSRF if attackers can obtain a victim's SID by performing session fixation or XSS. However, when attackers can obtain a victim's SID, attackers can masquerade as the victim without performing CSRF.

CSRF can also be prevented by checking the HTTP referer header, which holds the URI of the previous page from which a link was followed. A web application can only accept requests that come from valid domains or web pages by using the referer header. Since all referers of any requests issued by the attacker's script are in the attackers' domains or pages, web applications can distinguish requests. The web application must refuse requests in this approach with a whitelist of valid URLs. However, by setting visitors' browsers, visitors can send a request for which the referer header is discouraged from preventing information leaking to third parties. Since web applications cannot distinguish requests within this time, this approach cannot completely prevent CSRF attacks.

3. Automated Checking

3.1 Benefits of Automated Checking

Web application developers must check all kinds of vulnerabilities in the development phase of web applications. Unfortunately, this is almost impossible in practice for two reasons. First, web developers must be knowledgeable about all kinds of web vulnerabilities. Second, it is tedious to set up environments to check various vulnerabilities and prepare malicious links and/or scripts to emulate all kinds of attacks.

We automated the process of checking for session management vulnerabilities in web applications. We implemented our system as a plug-in of Amberate [6], [7], which is a framework for automated vulnerability scanners for web applications. Amberate provides a set of APIs to implement plug-ins to check for certain types of vulnerabilities in web applications. Amberate currently provides plug-ins to check for SQL injection, XSS, JavaScript hijacking, and directory traversal. Amberate is used in practice on several web sites such as the Open Government Lab [10] hosted by the Ministry of Economy, Trade and Industry of Japan.

This paper describes the design and implementation of Amberate plug-ins for session fixation and CSRF. The advantage of these plug-ins is twofold. First, by automating the process of checking for vulnerability, web developers do not have to have intimate knowledge of session management vulnerabilities. Second, the developers are released from the burden of having to prepare test environments to emulate malicious attacks since Amberate automatically mimics the malicious behaviors of attackers.

Amberate carries out real attacks on a target web application, and analyzes its responses to determine whether the attacks are successful or not. Each Amberate plug-in performs two tasks: 1) it generates messages to emulate certain kinds of malicious attacks and 2) it analyzes the responses to check for vulnerabilities. For example, a plug-in for session fixation generates attack messages to emulate session fixation, and it then analyzes the responses of the web application to check for vulnerabilities to session fixation.

To generate attack messages, Amberate occasionally needs application-specific information about messages exchanged between a browser and a web application. For example, Amberate requires a session name to specify an SID in an HTTP response's cookie issued by the web application and fix it to a victim in a session fixation attack.

Amberate consists of two phases: 1) a data-collection phase and 2) a vulnerability-check phase. A client of Amberate accesses a web application with his/her browser in the data capture phase as shown in **Fig. 3** in the first phase. For example, the client performs processes to log into the web application, post messages,

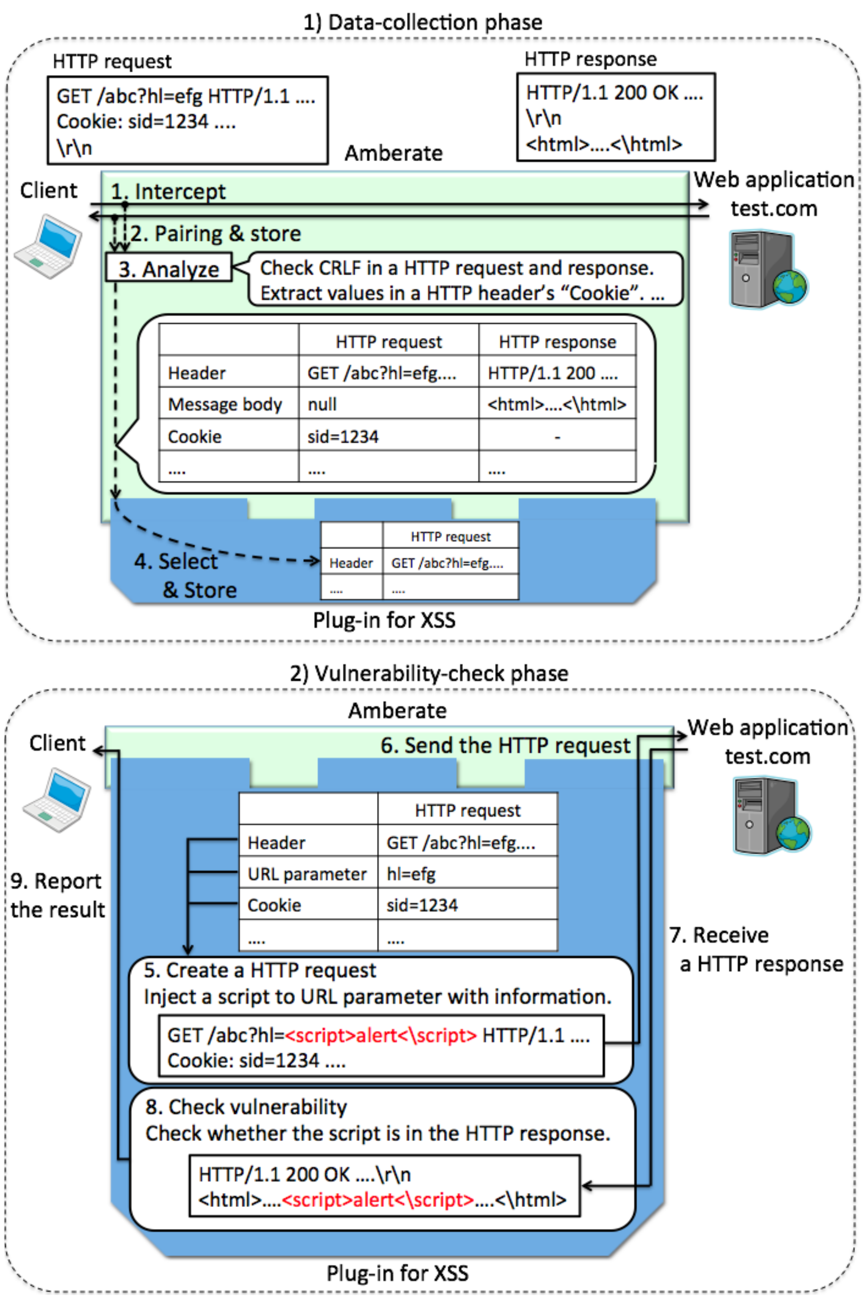


Fig. 3 Amberate.

and log out. Amberate works as an HTTP proxy and captures HTTP requests and responses between the browser and the web application (Step 1). Amberate pairs an HTTP request with an HTTP response and stores and analyzes them (Steps 2 & 3) at this time. For example, Amberate specifies the HTTP header and message body in the HTTP request and response by checking the CRLF in them. It extracts cookies in the HTTP header of the HTTP request by specifying variable names and values whose field name in HTTP header fields is "Cookie." It then extracts URL parameters in a URI of the HTTP request by specifying parameters that are between "?" in the URI and the end of the URI and parses the HTTP response's message body. Amberate provides this information including the HTTP request and response to plug-ins. All plug-ins can use this information provided by Amberate. The Amberate client does not need to perform processes to log in and out many times for all plug-ins. Each plug-in selects the necessary information for generating attack messages and stores it (Step 4).

Each plug-in generates attacks in the second phase with this information, sends them to the web application with Amberate functions, and receives HTTP responses and determines whether the web application is vulnerable or not by checking HTTP responses issued by the web application against an attack in the check phase. Methods of generating attacks and determining whether the web application is vulnerable or not differ according to the nature of the vulnerabilities. For example, a plug-in for XSS generates an HTTP request that injects a malicious script into the URL parameter and sends the HTTP request to a web application (Steps 5 & 6 & 7), as shown in Fig. 3. The plug-in uses information about the URL parameter provided by Amberate to generate the HTTP request. The plug-in for XSS checks whether the malicious script is in an HTTP response issued by the web application against the HTTP request, and it determines whether the web application is vulnerable to XSS and reports the result to the client (Steps 8 & 9). When the malicious script is in the HTTP response, the plug-in determines that the web application is vulnerable to XSS.

3.2 Plug-in for Session Fixation

A test operator can check for session fixation vulnerabilities by providing some application-specific information to Amberate. This application-specific information is the session name and the attacker's and victim's user names and passwords. We show examples of this information in **Table 1**. The test operator performs a login and a logout with his browser. Test operators can easily provide this information and log in and log out because Amberate users are the administrators or test operators of the target web application. Test operators do not set up test environments (two PCs, two browsers, and add-ons) and create messages or scripts for checking.

Our system automatically extracts the necessary information to mount a session fixation attack, e.g., a series of steps for login and logout, and a point that a user name and a password are embedded into in an HTTP request. It can extract this information from HTTP requests and responses provided by Amberate with information provided by test operators.

As we described in Section 2.2, the main problem with session fixation is that an SID that a web application has issued to a visitor can be used by other visitors. That is, if a target web application does not block other visitors (victims) from using an SID that has been issued to a visitor (an attacker), the target web application is vulnerable to session fixation.

Our system carries out the steps shown in **Fig. 4** to detect session fixation vulnerabilities. It logs into the target web application with the attacker's user name and password, logs out, and obtains an SID that the target web application has issued (Steps 1 & 2 & 3). Next, our system logs into the target web application with the victim's user name and password and the SID obtained in Steps 1, 2, and 3 (Steps 4 & 5 & 6). Finally, our system accesses the victim's page in the target web application with the SID obtained in Steps 1, 2, and 3 (Step 7).

Table 1 User inputs for phpBB to detect session fixation.

Application-specific information	Input example
Session name	phpbb2mysql_sid
Attacker's user name	attacker
Attacker's password	attacker_pwd
Victim's user name	victim
Victim's password	victim_pwd

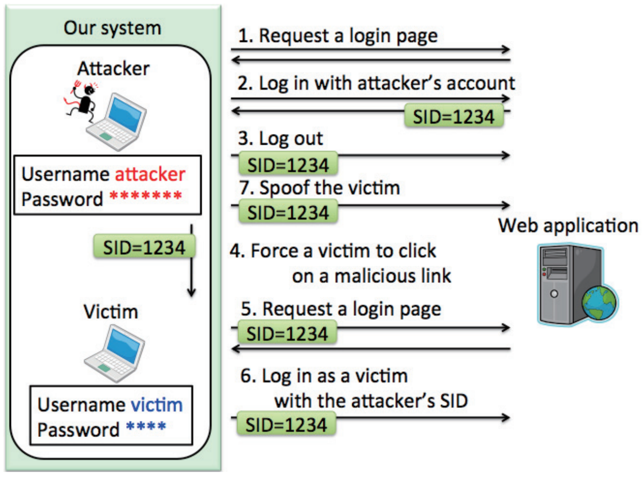


Fig. 4 Session fixation workflow.

Our system automatically extracts some information to perform these steps. A point that an SID is embedded into is required because our system obtains an SID in Step 2 and injects the SID into this point in Steps 3, 5, 6, and 7. A series of steps for login and logout are required because our system logs into and logs out of the target web application in Steps 1, 2, 3, 5, and 6. The login request and a point that a user name and a password are embedded into in an HTTP request are required because our system enters the attacker's and the victim's user names and passwords into the login request in Steps 2 and 6.

Our system automatically extracts this information from the HTTP requests and responses provided by Amberate with information provided by the test operator.

Our system specifies a point that an SID is embedded into by checking whether the session name provided by the test operator is embedded in a cookie in the HTTP response, the URI in the HTTP response, or a hidden field in the forms of the HTTP response. This is because the SID is embedded in the URI, the HTTP parameter, or cookie as we described in Section 2.1.

Suppose that the session name specified by test operators is "phpbb2mysql_sid." Our plug-in searches for a match to "phpbb2mysql_sid" in cookies, URIs, and hidden fields. If a match is found, for example, in a cookie in an HTTP response like "Set-cookie: phpbb2mysql_sid=6da54ea....; path=/," our plug-in concludes that the SID is embedded into a cookie.

Our system specifies the login request by analyzing HTTP requests and responses. We assumed that the value of the type attribute in the input element of the form that issued a login request would be "password." First, our system checks whether a request is a GET or POST request. If the request is a POST request, our system identifies which forms in the HTML document issued the request. Our system compares all variable names in the request message body with the values of the name attribute that the input elements of the form have. If variable names correspond to the values of the name attribute, the request is issued by this form. If the value of the type attribute that the input elements in the form have is "password," our system identifies that the request is a login request.

Suppose that a test operator issues a login request using the following form:

```
<form method="POST" action="./login_check.php">
  User name: <input type="text" name="username">
  Password: <input type="password" name="pwd">
  <input type="submit" value="Submit">
</form>
```

and issues the following request message body:

```
username=tester&pwd=*******
```

Receiving this request, our plug-in tries to specify which form issues this request. It retrieves and compares all the variable names in the request message body with those in each form. In this case, our plug-in retrieves variable names, `username` and `pwd`, and compares them with the above form. Since the variable names match each other, our plug-in determines this request is issued by the form. Then it finds out the form contains a variable whose type attribute is *password*, and concludes the above request is a login request.

Our system specifies points that a user name and a password are embedded into in an HTTP request by analyzing HTTP requests and responses with information on the login request. We assumed that an input form for the user name would be placed in front of an input form for the password in the login form. Our system checks input elements in the form that issues the login request from the above. First, our system stores the value of the name attribute in an input element to inject the user name into a login request and it checks the next input element. If the value of the type attribute in the next input element is not "password," our system updates the value of name attribute to inject the user name and it checks the next input element. If the value of the type attribute is "password," our system stores the value of the name attribute in the input element to inject the password. Our system determines the values which have the stored value of variable names in the login request's message body are points that a user name and a password are embedded into.

Suppose that our plug-in is trying to determine the variable name used for login names in the following form:

```
<form method="POST" action="./login_check.php">
   User name: <input type="text" name="username">
   Password: <input type="password" name="pwd">
   <input type="submit" value="Submit">
</form>
```

First, our plug-in stores `username` as a candidate and then checks the type attribute of the next variable (in this case, variable `pwd`). Since the type attribute of variable `pwd` is *password*, our plug-in concludes `username` is the variable used for login names.

Our system specifies a series of steps for login and logout by analyzing HTTP requests and responses with information on the login request. Our system determines a series of steps for login from a starting point in which test operators first send a request with their browser to a login point in which test operators obtain a response against the login request. It also determines a series of steps for logout from the login point to an end point from which test operators finally obtain a response.

There are ways for attackers to force the victim to use an SID. For example, when the target web application embeds an SID into a URI, an attacker creates a malicious link to the URI including an SID prepared by the attacker. The victim is forced to use the SID if he/she clicks on this malicious link. Test operators create this malicious link and check whether this link can force the victim to use the SID or not to mount a session fixation attack.

Our system does not create traps to force the victim to use an SID in Step 4 of the session fixation attack and in our system he/she directly uses the SID. The way the victim is forced to use the SID is not important, because the target web application cannot completely block the victim to force him/her from using the SID. It is important for testing session fixation that our system definitely forces the victim to use the SID and that it checks whether an SID that a web application has issued to a visitor (an attacker) can be used by other visitors (victims) or not.

Our system makes the test more effective before it mounts session fixation attacks. It checks whether an SID is changed at login or not. If the SID is changed at login, our system determines the target web applications are not vulnerable to session fixation since changes in the SIDs at login are an effective countermeasure. Our system extracts the SIDs from the session name provided by the test operator and HTTP requests and responses provided by Amberate. If the SIDs have changed when the test operator logs in, our system determines it is not vulnerable in this step.

3.3 Plug-in for CSRF

A test operator can check for CSRF vulnerabilities by providing some application-specific information to Amberate. This application-specific information is the session name, the victim's user name and password, the name of the secret token (if developers implemented a secret token mechanism as a countermeasure against CSRF), and the URI of a function that test operators would like to detect vulnerabilities. We show examples of this information in **Table 2**. Test operators perform a login, a function that test operators would like to check, and a logout with their browsers. Test operators can easily provide this information and log in and perform the function and log out. Test operators do not set up test environments (a PC, a browser, a server, and add-ons) and create messages or scripts for checking.

Our system automatically extracts necessary information to mount a CSRF attack. For example, a series of steps for login, and a point that a user name and a password are embedded into in an HTTP request. Our system can extract this information from HTTP requests and responses provided by Amberate with information provided by test operators.

As we described in Section 2.3, the main problem with CSRF is that a web application cannot distinguish requests that a visitor intended to send and those that he/she was forced to send by an attacker. That is, if a target web application does not block a request that an attacker forced a visitor (a victim) to send, the target web application is vulnerable to CSRF.

Our system performs the steps shown in **Fig. 5** to detect CSRF vulnerabilities in target web applications. It logs into the target web application with the victim's user name and password and obtains an SID that the target web application issued (Steps 1 & 2 & 3). Our system next creates an arbitrary HTTP request for the function in which test operators want to detect vulnerabilities and sends this request with the SID obtained in Steps 1, 2, and 3 (Step 4).

Our system automatically extracts some information to per-

Table 2 User inputs for phpBB to detect CSRF.

Application-specific information	Input example
Session name	phpbb2mysql_sid
Victim's user name	victim
Victim's password	victim_pwd
Name of the secret token	null
URI of a function	http://localhost/phpBB2/ privmsg.php?mode=post

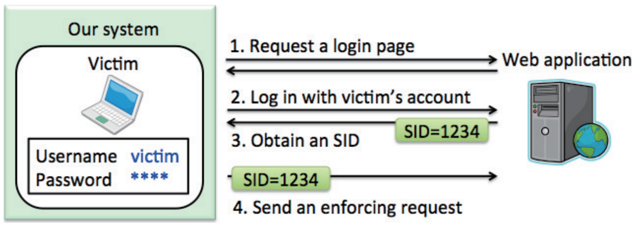


Fig. 5 CSRF workflow.

form these steps. A point that an SID is embedded into is required because our system obtains an SID in Step 3 and injects the SID to the point in Step 4. A series of steps for login is required because our system logs into the target web application in Steps 1 and 2. A login request and a point that a user name and a password are embedded into in an HTTP request are required because our system enters the victim's user name and password into the login request in Step 2.

Our system automatically extracts this information from HTTP requests and responses provided by Amberate with information provided by the test operator in the same way as it detects session fixation vulnerabilities.

Our system generates two or four types of requests which the victim is forced from the HTTP requests provided by Amberate and a function's URI where the test operator wants to detect vulnerabilities. Our system identifies a request to perform the function from HTTP requests with the URI and generates the four types of requests below from the request it identified.

(1) It does not have a referer header and a secret token.
(2) It has a referer header but does not have a secret token.
(3) It does not have a referer header but has a secret token.
(4) It has a referer header and a secret token.

Types 1 and 2 are generated because users can set up modern browsers to send empty or arbitrary values for this header for privacy. Types 3 and 4 are generated when a token mechanism is implemented in target web applications. If a target web application checks whether a request has a secret token and does not check the value of the token, the CSRF attack is successful when attackers force victims to send a request with a secret token generated by the attackers.

Our system makes the test more effective before it executes a CSRF attack. It checks the technique for embedding an SID. If the technique is URL rewriting or hidden field, our system determines the target web applications are not vulnerable to CSRF since these techniques are effective countermeasures. Our system extracts the technique from the HTTP requests and responses provided by Amberate. If the technique is URL rewriting or hidden field, our system determines that the web application is not vulnerable in this step.

3.4 Generating a Request

The attacker and the victim in our system send requests generated by our system in attacks. Our system generates the requests from HTTP requests provided by Amberate. Contents in a request are changed by user and time, etc. When our system obtains responses issued by the target web application in attacks, our system always monitors the responses. Our system modifies the HTTP requests provided by Amberate with the monitoring information. For example, the SID in the HTTP requests provided by Amberate is issued for the test operator. Our system needs to modify the test operator's SID in the HTTP requests to the attacker's and the victim's SID issued by the target web application.

3.5 Checking Attack Results

Our system checks whether an attack is successful or not by analyzing the response issued by the target web application in reply to an attack. If the response obtained by this attack is a response that the victim can obtain in session fixation, the attack is successful. If the response obtained by this attack in CSRF is a response obtained by treating the request that the victim was forced to send, the attack is successful.

Our system searches for a special keyword within the content of a response issued by the target web application in reply to an attack. If the special keyword appears in the HTTP document of a response issued in response to an attack, our system determines that this target web application is vulnerable. For example, the special keyword is "Welcome, victim (the victim's name)" and "Thank you, victim (the victim's name)." If the response issued in reply to a session fixation attack has the special keyword that appeared on a victim's page, this session fixation attack is successful. If the response in reply to a CSRF attack has the special keyword that only appeared when the target web application treated a message function and a purchase function, this CSRF attack is successful.

Our system requires test operators to provide a special keyword because it differs for each web application. False negatives and positives are created when our system extracts special keywords from the HTTP responses provided by Amberate.

4. Experiments

We evaluated our system using our synthetic and seven real-world web applications. We present and discuss our evaluations in this section.

4.1 Synthetic Web Applications

We first experimented with a synthetic web application vulnerable to session fixation or CSRF. We prepared seven types of session management mechanisms and implemented web pages for logging into and out of the web application and purchasing items. The seven types of session management are as follows. The first 4 mechanisms are related to session fixation and the last 3 are related to CSRF. Session management strategies 1 & 4 have session fixation vulnerability. Session management strategies 2 & 3 do not have session fixation vulnerability. Session management strategies 5 & 7 have CSRF vulnerability. Session management strategy 6 doesn't have CSRF vulnerability. We embed an SID in a request using three different approaches (cookie, URL rewriting, and hidden field).

(1) New SID issued on the first visit and new SID not changed at a user's login.
(2) New SID issued on the first visit and new SID changed at a user's login.
(3) New SID issued at a user's login.

Table 3 Detection results for the synthetic web application.
"No vul." means "not vulnerable to session fixation or CSRF." "-" means "not checked by Amberate because the target application is not vulnerable to CSRF."

Session managements	Vulnerability	Amberate		
		URL Rewriting	Cookie	Hidden Field
1	Session fixation	Session Fixation	Session fixation	Session fixation
2	No vul.	No vul.	No vul.	No vul.
3	No vul.	No vul.	No vul.	No vul.
4	Session fixation	Session fixation	Session fixation	Session fixation
5	CSRF	-	CSRF	-
6	No vul.	-	No vul.	-
7	CSRF	-	CSRF	-

Table 4 Detection results for session fixation.
"No vul." means "not vulnerable to session fixation." "-" means "not checked by Amberate because the target applications do not have a login function for administrators." "unable" means "cannot be checked with Amberate."

Web App.	User	Vulnerabilities	
		Manual	Amberate
Mambo 4.6.2	user	Session fixation	Session fixation
	admin	Session fixation	Session fixation
Joomla 1.0.9	user	Session fixation	Session fixation
	admin	Session fixation	Session fixation
phpBB 2.0.12	user	Session fixation	Session fixation
	admin	-	-
phpBB 3.0.9	user	No vul.	No vul.
	admin	No vul.	unable
phpNuke 7.0	user	No vul.	No vul.
	admin	No vul.	Session fixation
phpNuke 8.2.4	user	No vul.	No vul.
	admin	No vul.	Session fixation
osCommerce 2.2-MS1	user	Session fixation	Session fixation
	admin	-	-

(4) New SID issued only when a user does not have an SID at the user's login.
(5) Requests identified only with an SID.
(6) Requests identified with an SID and a secret token. The value of the secret token is checked to determine the requests are valid.
(7) Requests identified with an SID and a secret token. The value of the secret token is not checked.

Table 3 shows the experimental results. Amberate detects all the vulnerabilities in the synthetic web application. In the cases where there is no vulnerability in the web application, Amberate reports that the web application is not vulnerable.

4.2 Real World Web Applications

We conducted experiments against known vulnerable web applications in the real world after searching vulnerability repositories such as Refs. [11], [12], and [13]. The web applications we used were Mambo [8], Joomla [9], phpBB [14], phpNuke [15], and osCommerce [16]. We manually analyzed the source code of web applications and mounted session fixation and CSRF attacks to check vulnerabilities in web applications.

4.2.1 Session Fixation

Table 4 summarizes the names of seven real world web applications (Web App.), users who use the login function (User), results obtained from manual analysis and tests of vulnerabilities against session fixation (Manual), and results from our evaluations of session fixation (Amberate) in (Vulnerabilities). We can see two false positives occurred with two login functions and that our system cannot detect one vulnerability from this table. Our system could detect other login functions in web applications.

The reason our system caused the two false positives in phpNuke 7.0 and 8.2.4 is that the administrator's main page in phpNuke was the same as that of other administrators, and phpNuke did not blank out the SID when administrators logged out. We found a flow of false positives occurred when our system detected vulnerabilities in phpNuke.

(1) The test operator logs into phpNuke as an administrator and logs out with his browser.
(2) The test operator gives four items of information as a session name and the attacker's and the victim's user names and passwords, and a special keyword that appears on the victim's page, to our system. The special keyword is a string that appears on the attacker's and the victim's pages, because the administrator's main page in phpNuke is the same.
(3) Our system executes a session fixation attack with this information.
(4) The attacker logs into phpNuke, which issues an SID to the attacker.
(5) The attacker logs out. phpNuke does not blank out the attacker's SID at this time.
(6) The victim logs on with the attacker's SID. phpNuke issues a new SID to the victim at this time.
(7) The attacker accesses the administrator's page with the attacker's SID, which is valid.
(8) Our system determines that the session fixation attack was successful by analyzing the response obtained in the Step 7 with the special keyword obtained in Step 2. This is because the special keyword appears in the response.

phpNuke is not vulnerable to session fixation, but the method of session management that does not blank out the SID when administrators log out has problems. Developers should improve this method of session management. If attackers steal an SID from an administrator, they are able to use the SID after the administrator has logged out.

Our system cannot find a session fixation vulnerability in phpBB 3.0.9. This is because phpBB requests the administrator to input his user name and password twice to perform a login. Since our current implementation assumes that the user name and password are required only once, these pages cannot be checked. We

Table 5 Detection results for CSRF.
"No vul." means "not vulnerable to CSRF." "-" means "not checked by Amberate because the target application does not have functions for administrators." "unable" means "cannot be checked with Amberate."

Web App.	User	Exploit functions	Vulnerabilities	
			Manual	Amberate
phpBB 2.0.12	user	Send msgs	CSRF	CSRF
		Delete msgs	CSRF	CSRF
	admin	Modify admins' permission	No vul.	No vul.
		Modify users' info	No vul.	No vul.
phpBB 3.0.9	user	Send msgs	No vul.	No vul.
		Delete msgs	No vul.	No vul.
	admin	Modify users' info	No vul.	unable
phpNuke 7.0	user	add msgs	CSRF	CSRF
		delete msgs	CSRF	CSRF
	admin	modify users' info	CSRF	unable
		register users	CSRF	CSRF
phpNuke 8.2.4	user	add msgs	CSRF	CSRF
		delete msgs	CSRF	CSRF
	admin	modify users' info	CSRF	unable
		register users	CSRF	CSRF
Mambo 4.6.2	user	modify users' info	No vul.	No vul.
	admin	register users	CSRF	CSRF
Joomla 1.0.9	user	modify the user's info	No vul.	No vul.
	admin	register users	CSRF	CSRF
osCommerce 2.2-MS1	user	add goods in cart	CSRF	CSRF
		buy goods	No vul.	No vul.
	admin	-	-	-

believe a slight extension of our implementation can solve this problem.

Mambo, Joomla and phpNuke authenticate users with the SID and the IP address. Web applications can mitigate against session fixation attacks by authenticating users with them. Even if the attacker's and the victim's SIDs are the same by having forced the victim using the attacker's SID, the attacker's and the victim's IP addresses are not the same. A web application can distinguish the attacker from the victim and prevent session fixation attacks. However, this way of authenticating users with the SID and the IP address is not a good countermeasure against session fixation. If users access a web application with the same IP address in an intranet network and a session fixation attack occurs in this network, this cannot prevent session fixation attacks. Kolsek describes this way as mitigation [1]. Our system determines that the web application that authenticates users with the SID and the IP address is vulnerable to session fixation.

4.2.2 CSRF

Table 5 lists the names of seven real world web applications (Web App.), users who use the function (User), functions that we exploit to detect vulnerabilities (Exploit functions), results from manual analysis and detection of vulnerabilities against CSRF (Manual), and the results from our evaluation of CSRF (Amberate) in (Vulnerabilities). We can see from the table that our system cannot detect three vulnerabilities. However our system can detect other functions in web applications.

The reason our system cannot check the functions that modify user information in phpNuke 7.0 and 8.2.4 is that there is no special string on the page when user information is modified. When user information is modified, phpNuke does not forward users to the next page and sends them back to the first page. Consequently our system cannot carry out detection because test operators cannot give the string that appears in the response by handling the function that test operators want to test.

Our system cannot find a CSRF vulnerability in phpBB 3.0.9 in the same reason as the session fixation.

5. Related Work

Some web scanners are designed to check for session fixation or CSRF vulnerabilities. From the top 20 web vulnerability scanners at Insecure.Org [17] we compare Amberate with Wapiti [18] for session fixation and w3af [19] for CSRF. These two are chosen because they rank the highest among free and open-source scanners. To understand their mechanisms, we investigated the source code of these two scanners.

To summarize, these tools are less powerful than Amberate; they cannot detect the vulnerabilities used in Section 4. Wapiti is applicable only to a specific version of session fixation and thus, cannot detect all the session fixation vulnerabilities used in the experiments. w3af checks HTTP requests and responses only and misses the countermeasures implemented inside web applications. In the following, we show the detailed analysis and comparison of these tools.

Wapiti performs black-box scans on a target web application. It can detect some vulnerabilities such as session fixation, XSS, SQL Injection and HTTP Response Splitting. According to Wapiti's homepage [18] session fixation is categorized as CRLF injection. Attackers inject a CRLF and a malicious string into an HTTP response to split the HTTP response into two different responses (an original response and a malicious response). Attackers can force a victim's browser to handle a malicious response besides the original response. For example, attackers can force the victim to use an SID prepared by the attackers in the malicious response.

Wapiti can detect session fixation vulnerabilities only when there is the CRLF injection vulnerability. Unfortunately, session fixation is possible even if there is no CRLF injection vulnerability in web applications.

Amberate can detect session fixation even if it is caused by other security holes than CRLF injection. Because Amberate checks whether an SID that a web application has issued to a visitor can be used by other visitors or not. The main problem with session fixation is that a visitor's SID can be used by other visitors.

w3af is a web application attack and audit framework and can identify many web application vulnerabilities using plug-ins such as CSRF, XSS, SQL injection, and buffer overflow. In the w3af developers' mailing list [20], they discuss the CSRF plug-in and say "In most cases it will be better to *not use it (the CSRF plug-in)* because of a lot of false positive detection error." A CSRF plug-in only analyzes HTTP requests and responses which are audited to and from a target web application. It checks whether a request is a GET or POST request, the request has a query string or not, and the target web application uses at least one persistent cookie or not. It does not send a forced request for checking whether the target web application blocks the request or not.

w3af causes some false positives. The CSRF plug-in cannot check whether a countermeasure against CSRF in a target web application blocks CSRF or not because it does not check the contents of GET or POST requests. For example, secret tokens embedded in the HTTP requests are not checked in w3af. Amberate checks for target web applications by performing real attacks of CSRF and can check, for example, secret tokens.

Sania [7] and Secubat [21] automatically test web applications by attacking them. Sania intercepts requests and SQL queries between a web application and a database, and automatically generates SQL injection attacks from the information extracted from the SQL queries. SecuBat crawls web applications throughout the world to discover vulnerabilities against SQL injection and XSS by attempting real attacks. Sania and Secubat cannot be applied to detect session fixation or CSRF vulnerabilities because these vulnerabilities are not subject to input validation.

NoForge [22] and [23] prevent web applications with existing countermeasures against session fixation and CSRF attacks on the server side. One countermeasure for CSRF is to distinguish requests that visitors do or do not intend with a secret token and session fixation assigns a new SID each time a user logs in. NoForge is a technique where a server-side proxy detects and prevents CSRF attacks by using a secret token. Reference [23] is a technique where a server-side proxy detects and prevents session fixation attacks by using a second SID. Reference [23] issues a new second SID to a user and changes the second SID when the user logs in. NoForge [22] and [23] cannot detect session fixation or CSRF vulnerabilities in web applications because they check whether a request that a user has sent has been issued by session fixation and CSRF attacks or not.

The following work prevents session fixation and CSRF with new approaches. RequestRodeo [24] and BEAP [25] offer client-side protection from CSRF. Appisolation [26], SOMA [27], and Ref. [28] offer server and client protection from session fixation and CSRF. RequestRodeo can prevent CSRF attacks with a secret token in the browser. BEAP can prevent CSRF attacks with the concept where the browser limits the sending of sensitive data with a policy. Appisolation can prevent session fixation and CSRF attacks with the concept that the browser isolates sensitive data (e.g., cookies, and authentication tokens) per web applications and controls access to these web applications. SOMA can prevent CSRF attacks with the concept where the browser controls access to web applications with a policy issued by web sites. Reference [28] can prevent CSRF attacks with the concept where the browser issues a request including the Origin header and the server determines whether or not to handle the request with this Origin header. RequestRodeo [24], BEAP [25], Appisolation [26], SOMA [27], and Ref. [28] cannot detect session fixation or CSRF vulnerabilities in web applications because browsers check whether a request that users have sent has been issued by session fixation or CSRF attacks or not and they control the requests and sensitive data (e.g., cookies, and authentication tokens).

6. Conclusion

We presented a new system of automatically detecting session management vulnerabilities in web applications by executing real attacks in a simulated environment. The attacks were generated using some information about the target web application supplied by its test operators and the simulator automatically sent the attacks to the web application to check whether the attacks were successful or not. The test operators could easily detect these vulnerabilities by using this technique.

Our experiments demonstrated that our technique could detect vulnerabilities in our synthetic and some real-world web applications. We also discussed reasons our system could not check for vulnerabilities and provided details on false positives.

References

[1] Kolsek, M.: Session fixation vulnerability in web-based applications, available from ⟨http://www.acrossecurity.com/papers.html⟩.

[2] Shiflett, C.: Security Corner: Cross-Site Request Forgeries, available from ⟨http://shiflett.org/articles/cross-site-request-forgeries⟩.

[3] WhiteHat Security: WhiteHat Website Security Statistics Report, available from ⟨http://www.whitehatsec.com/home/resource/stats.html⟩.

[4] Bitcoin: available from ⟨http://bitcoin.org/⟩.

[5] Mt.Gox: available from ⟨https://mtgox.com/⟩.

[6] Kosuga, Y. and Kono, K.: Amberate: A Framework for Automated Vulnerability Scanners for Web Applications, *JSSST Trans. Comput. Softw.* (2011).

[7] Kosuga, Y., Kono, K., Hanaoka, M., Hishiyama, M. and Takahama, Y.: Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection, *Proc. Annual Computer Security Applications Conference* (*ACSAC '07*), pp.107–117 (2007).

[8] Mambo: available from ⟨http://www.mamboserver.com/⟩.

[9] Joomla: available from ⟨http://www.joomla.org/⟩.

[10] Open government lab: available from ⟨http://www.openlabs.go.jp/⟩.

[11] SecurityFocus: SecurityFocus, available from ⟨http://www.securityfocus.com/⟩.

[12] National Vulnerability Database: National Vulnerability Database, available from ⟨http://web.nvd.nist.gov/⟩.

[13] Cvedetails: available from ⟨http://www.cvedetails.com/⟩.

[14] phpBB: available from ⟨http://www.phpbb.com/⟩.

[15] phpNuke: available from ⟨http://phpnuke.org/⟩.

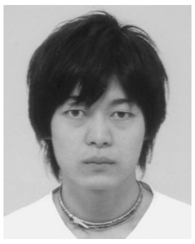
[16] osCommerce: available from ⟨http://www.oscommerce.com/⟩.

[17] Insecure.Org: available from ⟨http://insecure.org/⟩.

[18] Wapiti: available from ⟨http://wapiti.sourceforge.net/⟩.

[19] w3af: available from ⟨http://w3af.sourceforge.net/⟩.
[20] w3af-develop: available from ⟨http://sourceforge.net/mailarchive/forum.php?forum_name=w3af-develop⟩.
[21] Kals, S., Kirda, E., Kruegel, C. and Jovanovic, N.: SecuBat: A web vulnerability scanner, *Proc. International World Wide Web Conference* (*WWW '06*), pp.247–256 (2006).
[22] Jovanovic, N., Kirda, E. and Kruegel, C.: Preventing Cross Site Request Forgery Attacks, *Proc. Securecomm* (*Securecomm '06*), pp.1–10 (2006).
[23] Johns, M., Braun, B., Schrank, M. and Posegga, J.: Reliable protection against session fixation attacks, *Proc. 2011 ACM Symposium on Applied Computing* (*SAC '11*), pp.1531–1537 (2011).
[24] Johns, M. and Winter, J.: RequestRodeo: Client Side Protection against Session Riding, *Proc. OWASP Europe 2006 Conference, Refereed Papers Track, Report CW448*, pp.5–17 (2006).
[25] Mao, Z., Li, N. and Molloy, I.: Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection, *Proc. International Conference on Financial Cryptography and Data Security* (*FC '09*), pp.238–255 (2009).
[26] Chen, E.Y., Bau, J., Reis, C., Barth, A. and Jackson, C.: App isolation: Get the security of multiple browsers with just one, *Proc. 18th ACM Conference on Computer and Communications Security* (*CCS '11*), pp.227–238 (2011).
[27] Oda, T., Wurster, G., van Oorschot, P.C. and Somayaji, A.: SOMA: Mutual approval for included content in web pages, *Proc. 15th ACM Conference on Computer and Communications Security* (*CCS '08*), pp.89–98 (2008).
[28] Barth, A., Jackson, C. and Mitchell, J.C.: Robust defenses for cross-site request forgery, *Proc. 15th ACM Conference on Computer and Communications Security* (*CCS '08*), pp.75–88 (2008).

Yusuke Takamatsu received his B.E. and M.E. degrees from Keio University in 2010 and 2012, respectively. He is currently a Ph.D. student in Keio University. His research interest is web security. He is a student member of IPSJ.



Yuji Kosuga received his B.E. degree in 2007, M.E. degree in 2009, and Ph.D. degree in 2011 from Keio University. He is currently the Chief Technology Officier at Everforth Co., Ltd. His research interests are in web security and system software.

Kenji Kono received the B.Sc. degree in 1993, M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.