Session Fixation Vulnerability in Web-based Applications

Version 1.0

Mitja Kolšek *mitja.kolsek@acrossecurity.com*
ACROS Security *http://www.acrossecurity.com*

December 2002
Current copy available at http://www.acrossecurity.com/papers/session_fixation.pdf

1. Abstract

Many web-based applications employ some kind of session management to create a user-friendly environment. Sessions are stored on server and associated with respective users by session identifiers (IDs). Naturally, session IDs present an attractive target for attackers, who, by obtaining them, effectively hijack users' identities. Knowing that, web servers are employing techniques for protecting session IDs from three classes of attacks: interception, prediction and brute-force attacks. This paper reveals a fourth class of attacks against session IDs: session fixation attacks. In a session fixation attack, the attacker fixes the user's session ID before the user even logs into the target server, thereby eliminating the need to obtain the user's session ID afterwards. There are many ways for the attacker to perform a session fixation attack, depending on the session ID transport mechanism (URL arguments, hidden form fields, cookies) and the vulnerabilities available in the target system or its immediate environment. The paper provides detailed information about exploiting vulnerable systems as well as recommendations for protecting them against session fixation attacks.

2. Introduction

Web-based applications frequently use sessions to provide a friendly environment to their users. HTTP [1] is a stateless protocol, which means that it provides no integrated way for a web server to maintain states throughout user's subsequent requests. In order to overcome this problem, web servers – or sometimes web applications – implement various kinds of session management. The basic idea behind web session management is that the server generates a session identifier (ID) at some early point in user interaction, sends this ID to the user's browser and makes sure that this same ID will be sent back by the browser along with each subsequent request. Session IDs thereby become identification tokens for users, and servers can use them to maintain session data (e.g., variables) and create a session-like experience to the users.

There are three widely used methods for maintaining sessions in web environment: URL arguments, hidden form fields and cookies [2]. While each of them has its benefits and shortcomings, cookies have proven to be the most convenient and also the least insecure of the three. From security perspective, most – if not all - known attacks against cookie-based session maintenance schemes can also be used against URL- or hidden form fields-based schemes, while the converse is not true. This makes cookies the best choice security-wise.

Very often, session IDs are not only identification tokens, but also authenticators. This means that upon login, users are authenticated based on their credentials (e.g., usernames/passwords or digital certificates) and issued session IDs that will effectively serve as temporary static passwords for accessing their sessions.

This makes session IDs a very appealing target for attackers. In many cases, an attacker who manages to obtain a valid ID of user's session can use it to directly enter that session – often without arising user's suspicion. Interestingly, most cross-site scripting [3] proof-of-concept exploits focus on obtaining the session ID stored in browser's cookie storage. This class of attacks, where the attacker gains access to the user's session by obtaining his session ID, is called *session hijacking* [4].

Web session security is focused on preventing three types of attacks against session IDs: interception, prediction and brute-force attacks. Encrypted communication effectively protects against interception[1]. Using *cryptographically strong* pseudo-random number generators and carefully chosen seeds that don't leak from the server prevents prediction of session IDs. Finally, session IDs are immune to brute-force methods if their effective bit-length is large enough with respect to the number of simultaneous sessions[2].

Proposals have been made for mitigating the threat of stolen session IDs [6], and some products already implement such ideas (e.g., RSA Security's ACE/Agents for web servers).

3. Session fixation

As mentioned above, web session security is mainly focused on preventing the attacker from obtaining – either intercepting, predicting or brute-forcing - a session ID issued by the web server (also called "*target server*" in this paper) to the user's browser.

This approach, however, ignores one possibility: namely the possibility of the attacker "issuing" a session ID to the user's browser, thereby forcing the browser into using a chosen session. We'll call this class of attacks "*session fixation*" attacks, because the user's session ID has been fixed in advance instead of having been generated randomly at login time.

In a session fixation attack, the attacker fixes the user's session ID before the user even logs into the target server, thereby eliminating the need to obtain the user's session ID afterwards.

[1] *Although forgetting to mark session ID cookies as »secure« keeps the attacker's foot in the door [7].*

[2] *David Endler of iDefense wrote a very interesting article [5] on this topic.*

Let's take a look at a simple example of a session fixation attack. Figure 1 shows a web server `online.worldbank.dom` that hosts a session-aware web banking application. Session IDs are transported from browser to server within a URL argument `sessionid`.

First, the attacker – who in this case is also a legitimate user of the system – logs in to the server (**1**) and is issued a session ID `1234` (**2**). She then sends a hyperlink `http://online.worldbank.dom/login.jsp?sessionid=1234` to the user, trying to lure him into clicking on it (**3**). The user (how convenient for our example) clicks on the link, which opens the server's login page in his browser (**4**). Note that upon receipt of the request for `login.jsp?sessionid=1234`, the web application has established that a session already exists for this user and a new one need not be created. Finally, the user provides his credentials to the login script (**5**) and the server grants him access to his bank account. However, at this point, knowing the session ID, the attacker can also access the user's account via `account.jsp?sessionid=1234` (**6**).

Since the session has already been fixed before the user logged in, we say that *the user logged into the attacker's session*.

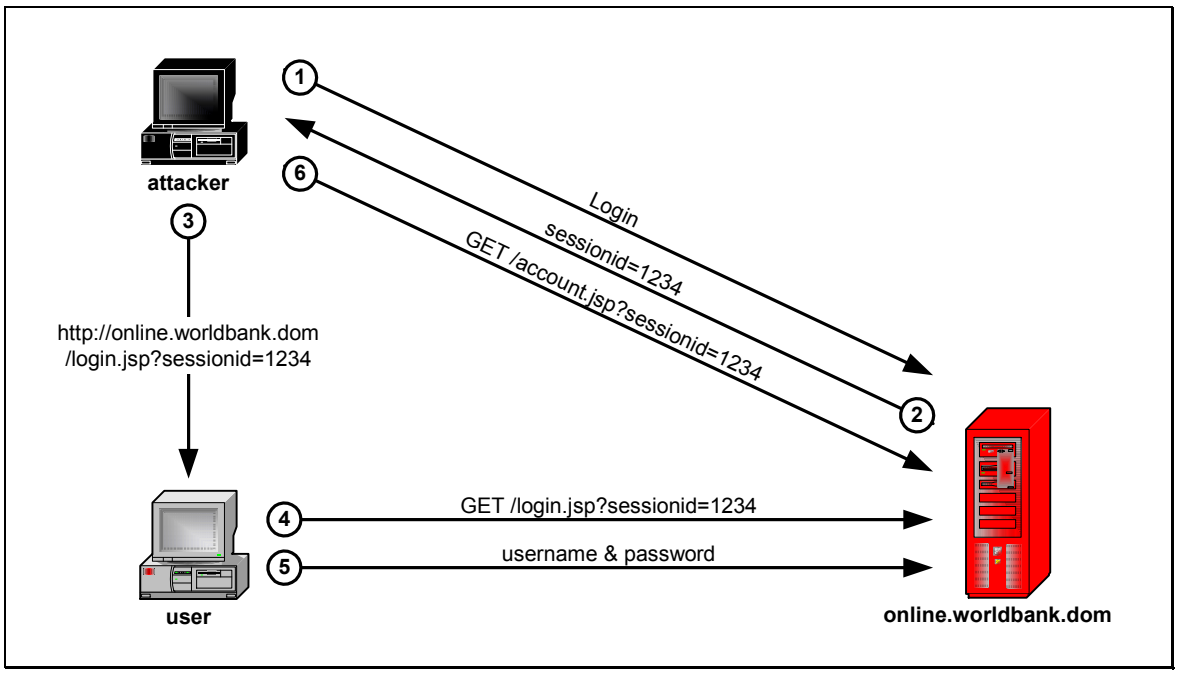

Figure 1: Simple session fixation in an URL-based web banking system

The above example is the simplest – and the least dangerous - form of a session fixation attack and has many shortcomings (for the attacker), such as: she has to be a legitimate user on the target server and she has to trick the user into logging in through the hyperlink she provided.

The following chapters will describe various methods for making session fixation more reliable, less detectable and available to "outside" attackers (those that are not

legitimate users on the target server). But first, let's examine the attack process step by step.

4. Attack process

Generally, session fixation attack is a three-step process, as shown in Figure 2:

1. **Session setup**: First, the attacker either sets up a so-called "*trap session*" on the target server and obtains that session's ID, or selects a – usually arbitrary – session ID to be used in the attack. In some cases, the established trap session needs to be **maintained** (kept alive) by repeatedly sending requests referencing it to avoid idle session timeout.
2. **Session fixation**: Next, the attacker needs to introduce her session ID to the user's browser, thereby fixing his session.
3. **Session entrance**: Finally, the attacker has to wait until the user logs in to the target server using the previously fixed session ID and then enter the user's session.

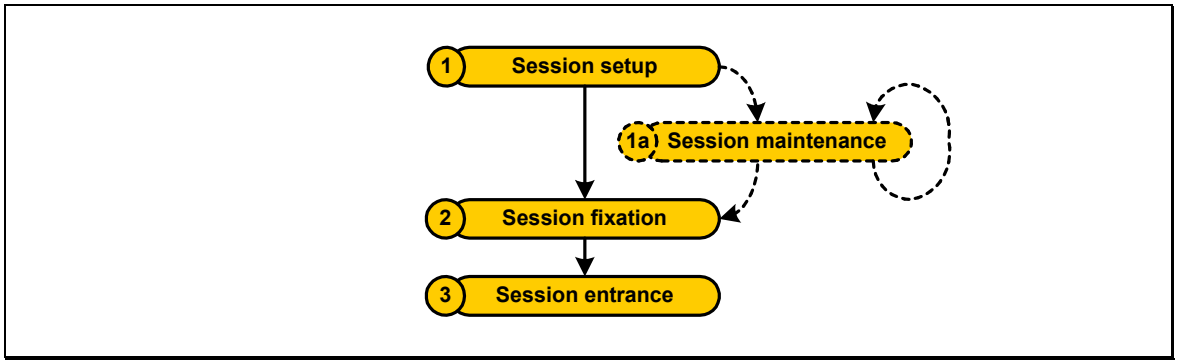

Figure 2: Three steps in a session fixation attack

Now let's take a look at the three steps in more detail.

STEP 1: Session setup

We can classify session management mechanisms on web servers in two classes:

a) "*Permissive*": those that accept arbitrary session IDs, and create a new session with proposed session ID if one doesn't exist yet (e.g., Macromedia JRun server, PHP).

b) "*Strict*": those that only accept known session IDs, which have been locally generated at some point in the past (e.g., Microsoft Internet Information Server).

For a *permissive* system, the attacker only needs to make up a random trap session ID, remember it and use it for the attack.

In a *strict* system, the attacker will have to actually establish a trap session with the target server, extract the trap session ID, remember it and use it for the attack. Depending on the session management logic, this session will need to be kept alive at

least until the user logs into it. Generally, sessions don't exist indefinitely; rather, servers automatically destroy them upon idle or absolute timeout. Idle timeout can easily be avoided by automatically sending periodic requests for the trap session. Absolute timeout, while less frequently employed, presents a tougher obstacle and in most cases defines the maximum timeframe for the entire attack process. Furthermore, restarting the web server can destroy all active sessions, requiring the attacker to return to the session setup step.

A *permissive* system requires no trap session maintenance.

STEP 2: Session fixation

In this step the attacker tries to transport the trap session ID to the user's browser. Methods for accomplishing that differ depending on the session ID transport mechanism. Let's examine each one of them.

Session ID in an URL argument

The attacker needs to trick the user into logging in to the target web server through the hyperlink she provides, for example,

`http://online.worldbank.dom/login.jsp?session=1234`.

This method, while feasible, is relatively impractical and comes with quite a risk of detection.

Session ID in a hidden form field

The attacker needs to trick the user into logging in to the target web server through a look-alike login form that in reality probably comes from another web server. This method is at least as impractical and detection-prone as the former one and is included here only for the sake of completeness. In the best case, the attacker could exploit a cross-site scripting vulnerability on the target web server in order to construct a login form (coming from the target server) containing a chosen session ID. However, the attacker managing to trick the user into logging in through a malicious login form could just as well direct the user's credentials to her own web server, which is generally a greater threat than that of fixing his session.

Session ID in a cookie

Cookies are a predominant session ID transport mechanism, partly also due to their security in comparison to URL arguments and hidden form fields. Ironically, on the other hand, cookies provide the most convenient, covert, effective and durable means of exploiting session fixation vulnerabilities.

What the attacker needs to do is install the trap session ID cookie on the user's browser. According to RFC2965 [2], the browser will only accept a cookie assigned either to the issuing server or the issuing server's domain. Consequently, even though it would provide a great attack avenue, the attacker's web server `www.attacker.com` *can't* set a cookie for the target web server `online.worldbank.dom`.

The attacker can choose among the three available methods for issuing a cookie to the browser:

A. using a client-side script that sets a cookie on the browser;
B. using the HTML `<META>` tag with `Set-Cookie` attribute;
C. using the `Set-Cookie` HTTP response header.

We'll analyze each of them in terms of their usability in a session fixation attack.

Method A: Issuing a cookie using a client-side script

Most browsers support client-side scripting, usually in JavaScript and/or VBScript languages. Both languages provide a way for the web server to issue a cookie to the browser by setting the property `document.cookie` to a desired value, for example in JavaScript:

```
document.cookie="sessionid=1234";
```

What the attacker wants to achieve is for the target web server to provide a client-side script like the one above that will issue the desired trap session ID cookie to the user's browser. This can be done using a well-known and very widespread vulnerability called the "cross-site scripting" [10].

Cross-site scripting

The attacker can exploit a cross-site scripting vulnerability on the `online.worldbank.dom` server in order to have that server issue the desired session ID cookie itself. An example URL for exploiting the cross-site scripting vulnerability in Microsoft's Internet Information Server [8] is:

```
http://online.worldbank.dom/<script>document.cookie="sessionid=1234";</script>.idc
```

The attacker could introduce this malicious URL to the user's browser by sending the user an HTML-formatted e-mail message including a small (invisible) frame originating from this URL. Alternatively, the attacker could send the user only a hyperlink to this URL (which can be disguised as something benign[3]) and try to lure the user into clicking on it.

Persistent cookies

The above cross-site scripting exploit can be enhanced by setting a persistent cookie instead of a temporary one (which is a default), thereby fixing the user's session for a longer period of time. Example:

```
http://online.worldbank.dom/<script>document.cookie="sessionid=1234;%20Expires=Friday,%201-Jan-2010%2000:00:00%20GMT";</script>.idc
```

Opening this URL in the user's browser will fix his session ID to `1234` until the year 2010. Unless the user clears his persistent cookies or reinstalls his computer, chances are that the attacker will gain a long-term access to the user's bank account. Clearly,

[3] *Microsoft Outlook, for example, doesn't make it easy for user to see where a hyperlink in a message is actually pointing to.*

this attack will be more effective on *permissive* systems, which don't require the attacker to maintain the trap session.

Domain cookies

Furthermore, domain cookies can expand the attack area from the target server to the entire target server's domain. Domain cookies are cookies with their `domain` attribute set to the issuing server's domain (e.g., "`.worldbank.dom`"). This attribute instructs the browser to not only send the cookie back to the issuing server but also to any other server in the specified domain. An example of exploiting a cross-site scripting vulnerability for issuing a domain cookie is this:

```
http://online.worldbank.dom/<script>document.cookie="sessionid=1234;domain=.worldbank.dom";</script>.idc
```

In a session fixation attack, the added value of domain cookies becomes clear in the following scenario (see Figure 3):

Attacker exploits a cross-site scripting vulnerability on server `www.worldbank.dom` (which is in the same domain as the target sever `online.worldbank.dom`) for generating a cookie-issuing script that sets a *domain* trap session ID cookie for the `.worldbank.dom` domain. She provides such cookie-issuing URL to the user's browser (**1**). Upon visiting the provided URL (**2**), the user's browser will accept a cookie from `www.worldbank.dom` (**3**) and will – since it's a domain cookie – send it to `online.worldbank.dom` when the user later decides to log in to his bank account (**4**).

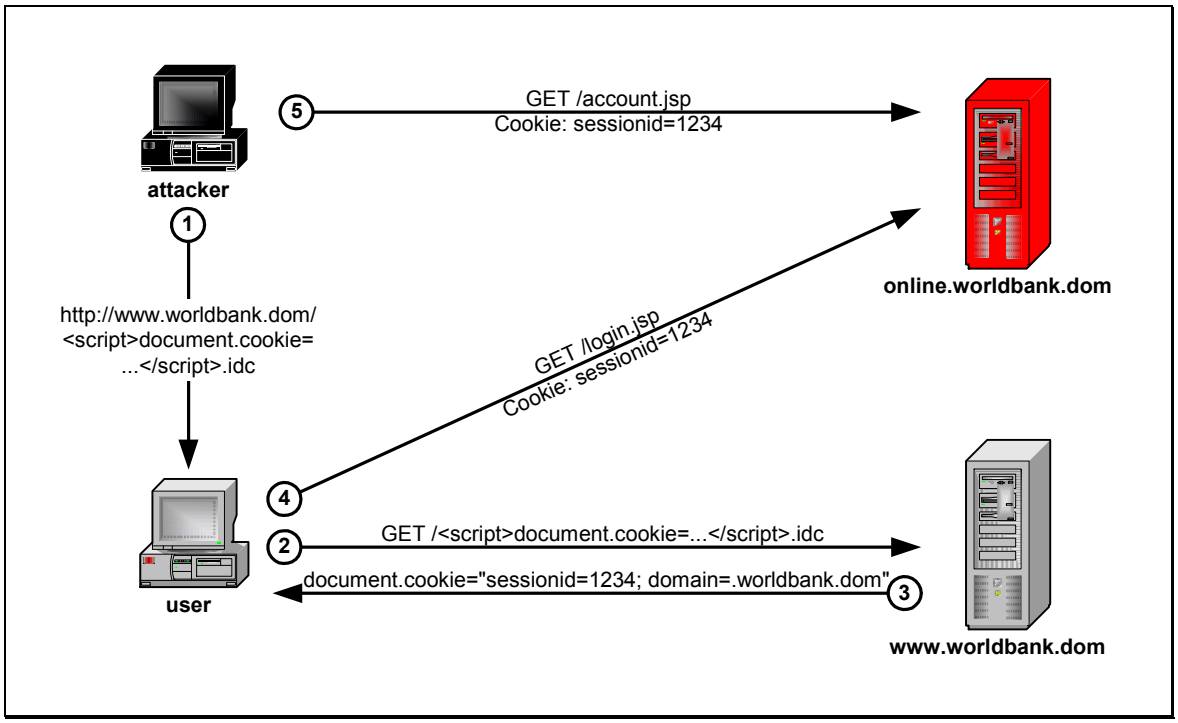

Figure 3: Session fixation using a cross-site scripting vulnerability on another server in domain

After the user's successful login, the attacker will be able to access his bank account using the fixed session ID (**5**).

A very effective attack scenario could employ a *persistent domain* trap session ID cookie.

Method B: Issuing a cookie using the <META> tag with Set-Cookie attribute

The server can also issue a cookie to the browser by including an appropriate `<META>` tag in the returned HTML document, for example:

```
<meta http-equiv=Set-Cookie content="sessionid=1234">
```

Now, the attacker wants the target server - or any other web server in the target server's domain - to return a specific, trap session ID cookie-issuing `<META>` tag to the user's browser. This can be achieved by using a "meta tag injection" method.

Meta tag injection

Most cross-site scripting vulnerabilities can also be used for injecting `<META>` tags to the resulting HTML document. Furthermore, systems that only scan the browser's arguments for `<script>` tag - which can effectively disable the introduction of unwanted scripts to the resulting HTML document - may still allow the injection of `<META>` tags. Note that although `<META>` tags are usually found between `<HEAD>` and `</HEAD>` tags, they are processed by browsers anywhere within the HTML document. An example of using the meta tag injection in a session fixation is this:

```
http://online.worldbank.dom/<meta%20http-equiv=Set-Cookie%20c
ontent="sessionid=1234;%20Expires=Friday,%201-Jan-2010%2000:0
0:00%20GMT">.idc
```

In contrast to client-side scripting[4], the processing of `<META>` tags can't be disabled on today's browsers[5], which makes the meta tag injection method superior to the client-side scripting method in a session fixation attack.

Method C: Issuing a cookie using the Set-Cookie HTTP response header

The third method for issuing a cookie to the browser is by including a `Set-Cookie` HTTP header in the web server's response. The attacker has a number of options to achieve that.

Session adoption

Some servers (e.g., JRun) accept any session ID in a URL and issue it back as a cookie to the browser. For example, requesting:

```
http://online.worldbank.dom/?jsessionid=1234
```

[4] *For example, client-side scripting is disabled in the Internet Explorer's Restricted sites zone.*

[5] *With the exception of the META REFRESH tag in Internet Explorer, which can be disabled.*

sets the session cookie `JSESSIONID` to `1234` and creates a new session with that ID on the server[6]. We'll call such behavior *"session adoption"*, due to the fact that the server effectively "adopts" a session ID that was generated by someone else.

Breaking into any host in the target server's domain

The attacker can try to break into any host in the target server's domain (for example, `wap.worldbank.dom`, `mail.worldbank.dom`, `forgotten.worldbank.dom`, etc.). Upon successful break-in, she can set up a simple cookie-issuing web server on that host, which will fix the user's session by issuing a domain trap session ID cookie to his browser.

Attacking the user's DNS server

As an alternative to exploiting a vulnerability on an already existing server in the target server's domain, the attacker can try to add her own web server to that domain by attacking the user's DNS server, as shown on Figure 4.

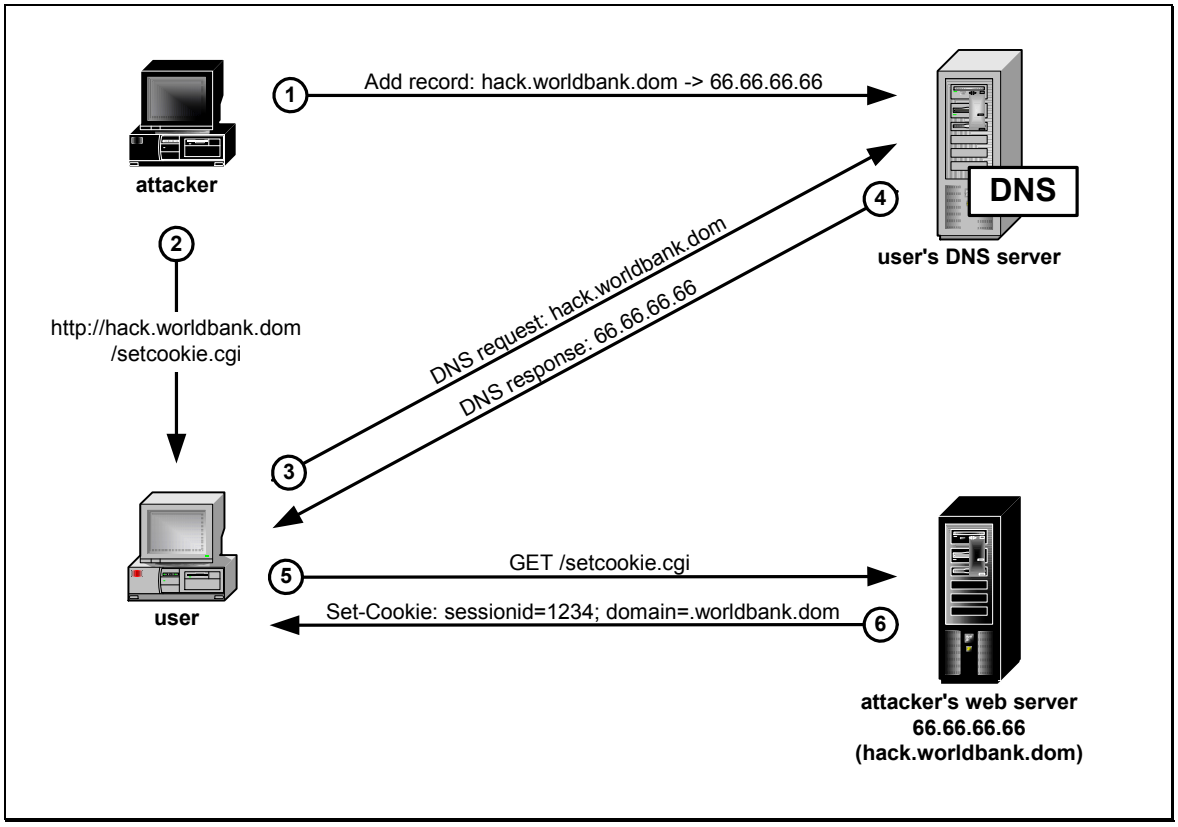

Figure 4: Fixing the user's session by attacking his DNS server

[6] *In JRun, it is required that a session with the proposed ID doesn't exist on the server yet in order for the server to return the session ID cookie. This may be different in other systems with similar behavior.*

First, she sets up a web server with a simple server-side script that issues a domain trap session ID cookie for the domain `.worldbank.dom`. She then adds a record to the user's DNS server (**1**)[7], mapping the hostname `hack.worldbank.dom` to her web server's IP address. To fix the user's session, she needs to make the user's browser request the cookie-issuing script on the `hack.worldbank.dom` server (**2**). The user's browser will ask the DNS server about the IP address of the `hack.worldbank.dom` server (**3**) and the DNS server will reply with the IP address of the attacker's web server (**4**). Thereafter, the browser will send a request for the cookie-issuing script to the attacker's web server (**5**) and will be issued a trap session ID cookie for the `.worldbank.dom` domain (**6**). Since the cookie will have been issued by a host in the `.worldbank.dom` domain, the browser will be forced to accept it, whereby the user's session would be fixed to the attacker's session ID.

Furthermore, the ability to add a host record to the user's DNS server also provides the attacker with an opportunity to introduce the malicious cookie-issuing URL to the user's browser in a more covert fashion (than via e-mail). By modifying one or more other records on the user's DNS server (e.g., for `www.yahoo.com`), she can redirect the user's browser to her web server when the user decides to visit the Yahoo pages. In this case, the fake `www.yahoo.com` server would first redirect the browser to the cookie-issuing script on the `hack.worldbank.dom` server and immediately thereafter redirect it to the real Yahoo web server to avoid suspicion.

Network-based attack

Finally, the attacker with ability to sniff and modify network traffic coming to and from the user's browser can also avoid sending the user a potentially suspicious e-mail and can perform the session fixation in an ultimately covert fashion. Namely, the attacker can inject a small (invisible) image in *any web server's* response to the browser – for example when the user is reading Yahoo news. This image would originate from any web server in the `.worldbank.dom` domain. Upon requesting the image content, the browser would connect to this web server and the attacker, intercepting the request, could send a fake response by the web server, including a `Set-Cookie` header, thereby fixing the user's session.

Notes: It's important to realize that using an encrypted communication between the user's browser and the target web server has literally no effect on the exploitability of session fixation vulnerabilities. It's also important to note that with cookie-based session mechanisms, the session fixation problem isn't solved automatically by fixing all cross-site scripting vulnerabilities on the target web server (which *can* solve the session hijacking problem, by the way). In fact, any vulnerable server in the target server's domain or a vulnerable DNS server can provide a session fixation opportunity for the attacker. For example, a seemingly irrelevant cross-site scripting vulnerability on one server can jeopardize a sensitive web server in the same domain. Finally, the `HttpOnly` cookie attribute [9], recently introduced by Microsoft mainly for the purpose of making session hijacking attacks more difficult, has little or no effect on session fixation attacks.

[7] *For example, using some DNS cache poisoning vulnerability.*

STEP 3: Session entrance

After the user has logged in to the trap session and before he has logged out, the attacker can enter the trap session and assume the user's identity. In many systems, the attacker will be able to use the session without the user noticing anything suspect. In case the user doesn't log out of the system, the attacker has an opportunity to keep the session alive – and thereby the access to the user's identity – for a long time.

Summarizing this section, Figure 5 graphically presents the session fixation process.

5. Countermeasures

First of all, we need to make it clear that preventing session fixation attacks is mainly the responsibility of the web application, and not the underlying web server. The web server – which usually provides the session management API to applications – should make sure that session IDs can't be intercepted, predicted or brute-forced. But as far as session fixation is concerned, only the web application can implement effective protection.

5.1. Preventing logins to a chosen session

There is one common denominator to all session fixation attacks and scenarios: the user logs in to a session with an attacker-chosen ID, instead of having been issued a newly generated session ID by the server. Since there seems to be no compelling reason for web applications to explicitly allow this to happen - and seems more like a side effect of current implementations –, we propose *forceful prevention of logging into a chosen session*. Web applications must ignore any session ID provided by the user's browser at login and must always generate a new session to which the user will log in if successfully authenticated.

5.2. Preventing the attacker from obtaining a valid session ID

If possible, a web application on a *strict* system should only issue session IDs of newly generated sessions to users *after* they have successfully authenticated (as opposed to issuing them along with the login form). This means that an attacker who isn't a legitimate user of the system will not be able to get a valid session ID and will therefore be unable to perform a session fixation attack.

5.3. Restricting the session ID usage

Most methods for mitigating the threat of stolen session IDs are also applicable to session fixation. Some of them are listed below.

- Binding the session ID to the browser's network address (as seen by the server)

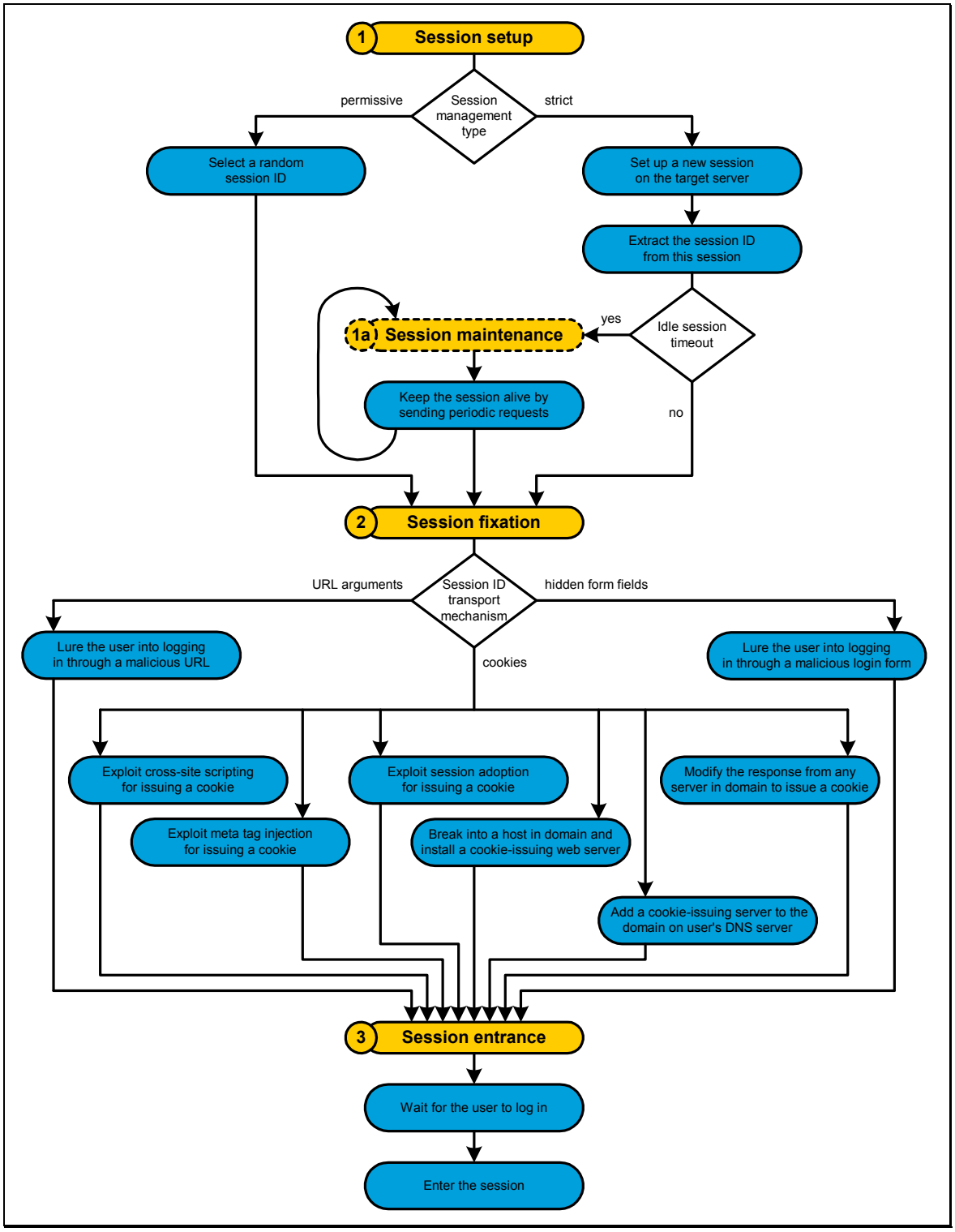

Figure 5: Session fixation process tree

- Binding the session ID to the user's SSL client certificate - very important and often overlooked issue in highly critical applications: *each* server-side script must first check whether the proposed session was actually established using the supplied certificate.
- Session destruction, either due to logging out or timeout, must take place on the server (deleting session), not just on the browser (deleting the session cookie).
- The user must have an option to log out – thereby destroying not just his current session, but also any previous sessions that may still exist (in order to prevent the attacker from using an old session the user forgot to log out from).
- Absolute session timeouts prevent attackers from both maintaining a trap session as well as maintaining an already entered user's session for a long period of time.

6. Conclusion

The session fixation vulnerability seems to be present in many session-enabled web-based applications. In fact, it has been present in almost all web-based systems (including many high profile web banking systems) that we've ever come across.

Understandably, we can't list any vulnerable real-world systems here but there are many out there - also among the largest ones. It would be an impossible job to try to make an exhaustive list of all publicly available vulnerable applications and inform their vendors about this vulnerability. Therefore we'll have to leave this job to these vendors and the worldwide security community in hope that as few as possible malicious exploits will see the light of day. And for all the guys out there doing the security reviews: this is just another checkbox to add to your checklist.

7. Session fixation vs. session hijacking

For reference, the following table presents the differences between session fixation and session hijacking vulnerabilities in terms of attack timing, impact duration, session maintenance, attack vectors and attack target area.

Timing	
Session fixation	Attacker attacks the user's browser *before* he logs in to the target server.
Session hijacking	Attacker attacks the user's browser *after* he logs in to the target server.
Impact Duration	
Session fixation	Attacker gains *one-time*, *temporary* or *long-term* access to the user's session(s).
Session hijacking	Attacker usually gains *one-time* access to the user's session and has to repeat the attack in order to gain access to another one.

Session Maintenance	
Session fixation	Can require the attacker to maintain the trap session until the user logs into it.
Session hijacking	Requires no session maintenance.
Attack Vectors	
Session fixation	1. Tricking the user to log in through a malicious hyperlink or a malicious login form 2. Exploiting a cross-site scripting vulnerability on any web server in the target server's domain 3. Exploiting a meta tag injection vulnerability on any web server in the target server's domain 4. Exploiting the "session adoption" feature of some web servers 5. Breaking into any host in the target server's domain 6. Adding a domain cookie-issuing server to the target server's domain in the user's DNS server 7. Network traffic modification
Session hijacking	1. Exploiting a cross-site scripting vulnerability on the target server 2. Obtaining the session ID in the HTTP `Referer` header sent to another web server 3. Network traffic sniffing (only works with an unencrypted link to the target server)
Attack Target Area	
Session fixation	Communication link, target web server, all hosts in target server's domain, user's DNS server
Session hijacking	Communication link, target web server

8. Acknowledgments

Many thanks to Kevin Fu of MIT, Saša Kos, Aljoša Ocepek and Stanka Šalamun for their invaluable contribution to this paper.

9. References

[1] IETF, »RFC2616: Hypertext Transfer Protocol -- HTTP/1.1«
http://www.ietf.org/rfc/rfc2616.txt

[2] IETF, »RFC2109: HTTP State Management Mechanism«
http://www.ietf.org/rfc/rfc2109.txt

[3] The Open Web Application Security Project, »Cross Site Scripting«
http://www.owasp.org/asac/input_validation/css.shtml

[4] The Open Web Application Security Project, »Session Hijacking«
http://www.owasp.org/asac/auth-session/hijack.shtml

[5] David Endler, »Brute-Force Exploitation of Web Application Session IDs«
http://online.securityfocus.com/data/library/SessionIDs.pdf

[6] Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster, »Dos and Don'ts of Client Authentication on the Web«
http://pdos.lcs.mit.edu/cookies/pubs/webauth:tr.pdf

[7] ACROS, »Remote Retrieval Of IIS Session Cookies From Web Browsers«
http://www.acrossecurity.com/aspr/ASPR-2000-07-22-1-PUB.txt

[8] SecurityFocus, »Microsoft IIS IDC Extension Cross Site Scripting Vulnerability«
http://online.securityfocus.com/bid/5900/info/

[9] Michael Howard, Microsoft, »Some Bad News and Some Good News«
http://msdn.microsoft.com/library/en-us/dncode/html/secure10102002.asp

[10] CERT, »CERT© Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests«
http://www.cert.org/advisories/CA-2000-02.html