Problems with Same Origin Policy
(Transcript of Discussion)

Dieter Gollmann
Hamburg University of Technology

Basic point, if you want to enforce the same origin policy, you have to be able to authenticate origin. In many cases, as you will see, one might be unable to do so for whatever reasons. But you might still be able to authenticate yourself, and that could be a useful security primitive, and that is one of the purposes of this talk, to discuss this security primitive of knowing yourself. I will use examples from web application security (which some of you might be much more familiar with than myself) to illustrate this point, and I could quite often refer to mobile network security, I see great similarities.

My first example of a problem is cross site scripting. Simple background: we have an attacker, we have a client, and we have a server trusted by the client. Attacker places malicious code somewhere on a webpage. If the webpage is directly at the server, this is known in the trade as stored cross site scripting; if the attacker somehow persuades the victim to submit this data himself to the server, this is known as reflected cross site scripting. When the server constructs a webpage, sends it back to the client, the malicious code the attacker had placed at the server comes back to the client, and is executed at the client side as coming from the trusted server, so it’s running with the permissions of the trusted server. The attacker somehow manages to create an interesting webpage at the attacker site, the victim clicks on something, something is transported from the attacker to the victim containing the malicious code, the victim might add more information, then sends it to the trusted server, the trusted server sends back a response and the malicious code will now be executed on the client as coming from there, and not from here, and that’s why it’s called cross site scripting.

Ultimate cause, fundamental problem: failure of origin based policy. Those who know me, know authentication is a topic I’ve been beating up on for a very long time. We’ve only authenticated the entire page, we have not been looking at the bits and pieces in the page, and in this case some of those pieces came from the attacker.

The first defence I would classify as saying, OK, origin based policy doesn’t work here, the problem really is we’re executing code when we don’t like to execute code, therefore let’s try to differentiate between code and data. Very good. The server might sanitise the output it sends out. We might use escaping so the dangerous characters that initialise the script, or start the script, are somehow blocked. I am tempted to call this a band-aid because it works in many cases sort of reasonably well, but not always. And I have an example, which I hope some of you might find interesting and educating, and might raise
general discussions about abstraction security. It’s admittedly slightly tangential to my topic, it’s about SQL injection attacks.

There is a defence against SQL injection called the `addslashes` function, which adds a slash in front of every dangerous character to escape that dangerous character. And the other part of the story is the GBK character set for simplified Chinese. The story I found on this website from Chris Shiflett[1] as part of a discussion. What he observed is that hexadecimal character BF27 is not a character in GBK. If we look at it byte by byte, it’s BF and 27 and 27 is a single quote, and single quote starts a string, so it’s a dangerous character, and we have to escape it. So we add a slash which is 5C in hexadecimal representation. However, what we get then is BF5C which happens to be in the GBK character set, so we have this character, followed by a single quote, despite the fact that we’ve just tried to put a slash in front of the single quote. So something very interesting is happening here with abstractions. You have the abstractions in the character set, you have the byte level representation, and somehow inserting slashes at the byte level doesn’t insert slashes at the level of the abstraction. So that is something to think about.

Second defence, which Martin Johns called session-safe. The idea is a simple one. At the time the server starts a session with the client, the server sends unpredictable one time URLs. The server is now able to recognise its own URLs because it has created those URLs in some unpredictable way, and if it has kept track of who has received those URLs, it knows who has been sending these requests. So in the cross site scripting example an attacker would have to include a URL in the request to the server. If it’s an unpredictable URL that had not been compromised at the client, the attacker would not be able to do this, and in this way we would get authentication of origin, courtesy of this know thyself primitive.

Next example in my list: cross site reference. In essence you see the same players involved, the same flow of information. The main difference: in this case the user has an authenticated session with the server, the server is the target system, so the request coming from the attacker will be treated by the target as a request coming from the user, and again you have a violation of an origin based policy. Exploit the trust, trust the target website has with the user. The user is authenticated, be it through cookies, be it through an authenticated SSL session, or something else. And again, the user has to go to the attacker’s website to start the attack. When the user goes to the attacker’s website the client browser will submit the attacker’s data to the server, and the server will treat this as coming from my authenticated user. Violation of the target’s origin based security policy, and the same story as before. Ultimate cause: the server only authenticates the entire page, or the entire request, I should have said, and not the individual parts in this request.

In the literature I see two types of defences, one is a server initiated defence. Here the interesting thing is that, to authenticate the request as coming from a particular user, or a particular client, you go up above the level of the browser to

[1] shiflett.org