Protocol for Mitigating the Risk of Hijacking Social Networking Sites

Jeffrey Cashion and Mostafa Bassiouni
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Florida USA
jcashion@knights.ucf.edu, bassi@cs.ucf.edu

Abstract- **The proliferation of social and collaborative media has been accompanied by an increased level of cyber attacks on social networking and collaboration sites. One serious type of attack is session hijacking attacks which enable the attacker to impersonate the victim and take over his/her networking session(s). In this paper, we present a security authentication protocol for mitigating the risk of hijacking social networking and collaboration sites. The protocol is based on the recognition that users of social and collaborative media connect to their websites using a variety of platforms and connection speeds. To appeal to both mobile devices such as smart phones or tablets using Wi-Fi connections and high-end workstations such as PC's using high-speed connections, a novel *Self-Configuring Repeatable Hash Chains* (SCRHC) protocol was developed to prevent the hijacking of session cookies. The protocol supports three different levels of caching, giving the user the ability to forfeit storage space for increased performance and reduced workload. Performance evaluation tests are presented to show the effectiveness and flexibility of the SCRHC protocol.**

Index Terms - **Social Networks, Session Cookies, Session Hijacking, Security Protocols**

I. INTRODUCTION

The growth of social media and the increase in the number of users of social networking sites (SNSs) in the past few years are mind-boggling. Initially, social media has been used by ordinary people just for connecting with friends and for making new friends. A large population of people worldwide are now acclimated to social networking and the use of modern technology (e.g., smart phones, tablets, PDAs) to communicate with friends and co-workers. Social media has recently started taking important role in business as well. Companies have started using social media websites such as Twitter and Facebook for doing marketing, market research and customer support. The proliferation of social media has, however, been accompanied by a similar level of growth in cyber attacks on social networking sites. In addition to phishing and spamming attacks, threats to SNSs include session hijacking attacks that enable the attackers to view private photos, broadcast messages, see personal web history, and do anything else that the owner of the hijacked account can do. The threat of weak security to a SNS could hurt its adoption and scare away future users from engaging in the site any more than they already do. For this reason, SNS owners should take a serious look at this issue and seek to adopt a solution that is both efficient and elegant.

In this paper, we investigate the problem of session hijacking of social media and propose a protocol for combating this type of attack. We present multiple flavors of our protocol which are suited for a variety of client platforms as well as connection speeds. The flexibility of the protocol allows the client and server to be configured to suit each user's own personal preferences. Flexibility also permits a website security administrator to selectively offer the service provided by the protocol and adjust the amount of resources he is able to dedicate, given his current server ability. This makes it easier for the protocol to gain acceptance.

COLLABORATECOM 2011, October 15-18, Orlando, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2011.247167

II. SECURITY RISKS IN SOCIAL NETWORKS

There has been a recent surge in the number of research reports and security-related blogs that alert the Internet community to the serious security risks facing the users of social networks. We discuss some of these reports below.

A novel friend-in-the-middle attack (FIMA) on social networks has been revealed in February 2011 [7]. The FIMA is basically an active eavesdropping attack based on the missing protection of the communication link between users and social networking providers. Session cookies of many social media sites are saved locally at the client side. These cookies contain among other information a shared hashed secret, which is used as a proof that the user has been successfully authenticated. As these cookies are transmitted unencrypted, the communication between a user and the social media provider is vulnerable to cookie hijacking. Thus, an attacker could take over a user's social networking sessions by sniffing out the HTTP cookies, since the majority of social network providers do not support HTTPS. By hijacking session cookies, it becomes possible to impersonate the victim and interact with the social network without proper authorization. Huber et al. [7] showed that a friend-in-the-middle attack can be used for context-aware spam and social phishing on a large scale. The study in [7] presented an evaluation of the feasibility of this attack on Facebook; the report also noted that Facebook plans to offer optional HTTPS support for their web service and advised users to make use of this option once it will become available to everyone.

In May 2011, Rosario Valotta [2] revealed an unpatched vulnerability in all versions of Internet Explorer (IE) that can be exploited to hijack people's online identities. The attack tactic, dubbed cookie jacking, exploits a 0-day vulnerability affecting every IE version on every Windows OS box installation. The attack leverages on a User Interface redressing approach and allows an attacker to steal session cookies from any social media site a victim is visiting. The 0-day attack can be explained as follows. IE defines Security Zones as a proprietary mechanism that allows users to group web sites according to their source's trust. An attempt to access an *iframe* source stored in a more-privileged zone (e.g., local file on a PC) from a less-privileged zone (e.g., Internet zone) will result in an Access Denied error. However if the iframe source is set to a cookie file, the iframe will successfully load the content. This is a 0-day vulnerability that results in iframe loading the cookie and, as claimed in [2], works across any IE version on any Windows OS box.

In October 2010, security programmer E. Butler released a free open source Firefox extension, called Firesheep [3], to demonstrate the vulnerability of public Wi-Fi and Web 2.0 applications to cookie-sniffing and to raise awareness about the dangers of cookie hijacking. The design of the Firesheep software is based on the observation that it is common for web sites to protect user’s password by encrypting the initial login only, and not encrypting anything else. HTTP session hijacking is when an attacker gets a hold of a user's cookie, allowing them to do anything the user can do on a particular web site. The ultimate goal of the Firesheep software is to put pressure on service providers in order to adopt more rigorous security policies and offer robust authentication protocols to protect the people who depend on their services. Concurrently Butler and Gallagher [4] reported that social networking sites and many companies including Facebook, Twitter, and even Google all fail to protect users against session hijacking attacks. They demonstrated this by releasing an open source tool which shows a “buddy list” of people’s online accounts being used around the attacker; the attacker simply double clicks to hijack any selected user.

A report posted in May 2011 on the personal weblog of security researcher Rishi Narang [5] has generated considerable attention. The report shows that cookies of the social networking LinkedIn site may be active for up to a year. After the login process, LinkedIn creates a file on the user’s computer which the site then uses for

quicker access later on, just like cookies on many other sites. However, the extended expiry time for LinkedIn means a bigger window of opportunity for cyber criminals. If a hacker can access the relevant file, they can continually access a user's account for extended time.

III. PREVIOUS WORK

In [6], we presented a protocol to prevent cookie hijacking in wireless networks. The protocol, called Rolling Code, utilizes the initial secure HTTPS authentication to exchange a shared secret between the server and the user browser. The shared secret consists of two components: a *seed* and value *d*, both of length 160 bits. For every transmission made from the client to the server, the client first updates the value of d by hashing it then generates a cookie code which is another hash operation applied on the XOR of seed and the updated value of d. The client sends the cookie code to the server which will perform similar steps on the shared secret stored at the server and compare the computed cookie code with the received cookie code. A number of other protocols prior to the Rolling Code protocol were proposed in the literature. We briefly outline some of the relevant protocols below.

Liu et al. [8] proposed a secure cookie protocol by making modifications to improve an earlier protocol proposed by K. Fu [9]. Their solution for ensuring integrity of each cookie involved embedding a username, expiration, data, and HMAC. This would inject a lot of repetitive data if there were a lot of cookies; each cookie would all have this information embedded in it. Also, it is not confidentially protecting the names of the cookies, but instead leaving them open for all to see what kinds of information is being shared. Their fundamental assumption is that their secure cookie protocol would run on top of SSL, which is an expensive protocol in terms of its computational overhead.

Recently, work has been done to establish formal guidelines for a one-time-use cookie authentication token [10] using hash chains. The use of a hash-chain requires the client and server to establish how many transactions they expect to do during the lifetime of the connection. Such a value is expected to be estimated by the website administrator ahead of time using metrics based on usage statistics. This poses an obstacle when trying to achieve a solution with minimal overhead.

In addition to protocols that combat hijacking attacks, there has been some work to develop software that helps the user avoids sites that do not provide adequate security and pose a threat to user privacy. An example of this software is a recent plug-in for Firefox that can notify users of such a threat [11].

IV. AUTHENTICATION PROTOCOL

In this section, we extend our work in [6] and present a security authentication protocol for mitigating the risk of hijacking social networking sites. It is important to stress that our protocol is only intended to prevent attacks related to cookie hijacking in social networks. Specifically, our protocol cannot be used to combat a myriad of other serious attacks including worm-based attacks. One example of attacks that cannot be handled by our protocol is the nefarious Koobface worm which has repeatedly targeted users of social networking websites such as Facebook, MySpace, and Twitter. Koobface, whose name is Facebook scrambled, is used by cybercreeps to hijack social media accounts without hijacking session cookies. Basically, the Koobface attack arrives in the form of a message from a friend asking to download a special video player to view a video. The download triggers an automated program that sends copies of the same viral message to all of the victim's friends, while turning full control of the victim's PC over to the attacker.

Our protocol is based on the recognition that users of social media such as Facebook connect to their web sites using a variety of platforms. On one end, there are Wi-Fi connections with users connecting via mobile smart phones or tablets. On the other end, there are high-speed

connections with users connecting via high-end PC's and workstations. We therefore employ two different authentication flavors: one for mobile devices using wireless connections and the other for high-end workstations using high-speed broadband connections. The core of the two flavors is the same, but they differ in how they exercise various aspects of the protocol.

For devices of all types and connections, we modify the hash chains approach in order to overcome a known limitation regarding the need to estimate the number of transactions during the lifetime of a session. We call this modification the *Self-Configuring Repeatable Hash Chains* (SCRHC) Protocol. Below, we provide motivation for SCRHC then present its basic design.

The hash chains approach has been used in the one-time-cookies (OTC) authentication protocol [10] to prevent session hijacking. The use of a hash-chain requires the client and server to establish how many transactions they expect to do during the lifetime of the session. Such a value is expected to be estimated by the website administrator ahead of time using metrics based on usage statistics. If the number of transactions is overestimated, the authentication in the early steps will suffer from an unjustified large computational overhead. If the number of transactions is underestimated, there will be the undesirable synchronization overhead of establishing a new secret and a new number for the remaining transactions.

The hash chains approach can be formally described as follows. In the mth transaction of the session, the value of the authentication Code to be transmitted from the client to the sever will depend on the value of the initial secret s, the estimated number of transactions in the session n, and the value of the transaction index m as follows

$m = 1 \rightarrow Code = H^{n}(s)$ // 1st transaction
$m = 2 \rightarrow Code = H^{n-1}(s)$ // 2nd transaction
…
$m = j \rightarrow Code = H^{n-j+1}(s)$ // j^{th} transaction

Notice that in the first transaction, the browser performs the hash operation n times in order to obtain $H^{n}(s)$.

Let n be the number of transactions per session, which is not known in advance. Rather than trying to estimate the value of n, the SCRHC protocol uses a relatively small value *Gk*, set to 10 in our testing, which we call the *base chain length*. The actual chain length, *k*, is set to *Gk* at first. If n is greater than *k*, both the client and server execute a routine, called Repeat_Chain, denoted RC, to compute a new value for *k* without exchanging any messages or invoking new HTTPS authentication. This modification is done by utilizing *Gk* and another value *r*, which is initially set to one, and increases each time RC is called. The value *k* is assigned the value of *Gk* multiplied by *r*. For example, the first time that RC is called, *r* is incremented to 2, and *k* is assigned the value $Gk \times 2$. The RC routine also changes the value of the secret *s* to prevent repeat-attacks.

Our protocol supports three different levels of caching, giving the user the ability to forfeit storage space for increased speed and reduced work-load. The simplest method is no caching. For each communication event, the SCRHC_Step function must generate the code in its entirety from scratch. This is the most time consuming but also does not require any storage. There are situations where storage space might simply not be available or in limited supply, so we allow for this.

The second level of caching is what we call *selective dynamic caching*. When the value *k* is first set at initialization or when it is updated via the Repeat_Chain function, we then decide how many hash values we will store in cache. This amount is set to $\sqrt{k}$, spaced equally apart. For example, if the value of *k* is set to 100, then our cache would contain values for s hashed iteratively the following number of times: {0, 10, 20, 30, 40, 50, 60, 70, 80, 90}. For each SCRHC_Step, the closest cached hash value of s is fetched, and the remainder of the hashing is done. For example, if *k* was initially valued at 100 and is currently valued at 63, then $hash^{60}(s)$ is fetched from cache, and then hashed 3 more

times to get the value $hash^{63}(s)$. This is clearly much faster than performing the hash 63 times. Our testing shows that this technique provides approximately a 10x speedup when compared to SCRHC (no caching) while only requiring the storage of a dozen or so values in cache.

The third level of caching is full caching, where all of the hashes are computed ahead of time and stored for later reference. This requires the user to dedicate considerably more storage than either of the other two methods, but if this storage is available, it provides the fastest performance. Our technique requires much less space than OTC though, since our initial chain lengths are short, thus requiring fewer cached hashes. As will be shown in section V, our protocol requires around 1/10th the cache storage of OTC for any typical session.

A. Configuration for Mobile Devices

For mobile devices, we recommend that the client dedicate as much space as they can to caching to reduce the computational overhead. In some cases, however, it is understood that the device might have very limited storage capabilities, such as in embedded devices. In this case, the device should at least try to opt for our selective dynamic caching.

B. Configuration for High-end Workstations

For more powerful devices such as desktop computers, we recommend that the client run full-caching. In a typical desktop, RAM is overly abundant, so storing every hash in cache is not such a wild idea. The amount of memory required for full caching for SCRHC using Gk = 10 (default value) for a session of 2,500 communications is roughly 5KB.

C. High-level Pseudo Code of SCRHC Protocol

The following is high-level pseudo code of the SCRHC protocol. We include all three forms of caching for clarity, but only one would be used at a time, and the client and server can each use their own level of caching.

Initialization:
The initial value of the shared secret *s* and the base chain length *Gk* are selected and exchanged between the server and the client during the initial HTTPS authentication. The cache is filled by the fillCache function, if required.

```
k := Gk     // k is initially set to Gk × 1
r := 1      // r is the current chain number
Call fillCache()
```

Filling Cache:
This fills the cache, if it is being used.

fillCache *(No Caching)*:

```
return  //don't do anything. No caching required.
```

fillCache *(Selective Dynamic Caching)*:

```
miniCacheInterval := √k                //square root of k
miniCacheK := k - miniCacheInterval  //highest item
miniCacheIndex := cache.size - 1  //point to last item
cache[0] = s;
For i := 1 to miniCacheIndex Do
   cache[i] := hash^miniCacheInterval(cache[i-1])
End-For
```

For example, if k is 100, then miniCacheInterval = 10, miniCacheK = 90, and miniCacheIndex = 9.
The For-Loop essentially achieves the following:
cache[0] = $hash^{0}(s)$, cache[1] = $hash^{10}(s)$,
cache[2] = $hash^{20}(s)$, cache[3] = $hash^{30}(s)$,
cache[4] = $hash^{40}(s)$, cache[5] = $hash^{50}(s)$,
cache[6] = $hash^{60}(s)$, cache[7] = $hash^{70}(s)$,
cache[8] = $hash^{80}(s)$, and cache[9] = $hash^{90}(s)$.

The actual code is slightly more complex than this, but the end result is equivalent.

fillCache *(Full Caching)*:

```
cache[0] :=  s
//Perform the hash operation k times to obtain H^k(s)
For i := 1 to k Do
   cache[i] :=  hash(cache[i-1])
End-For
```

Updating:

For every transmission (step) made from the client to the server, the client will perform one of the following code segments, depending on the level of caching used.

```
SCRHC_Step (No Caching):
    Code := s
    //Perform the hash operation k times to obtain H^k(s)
    For i := 1 to k Do
       Code := hash(Code)
    End-For
    k := k - 1  // decrement  k
    //Compute new values for the next transaction
    If (k == 0) Then Call Repeat_Chain(s, G_k, r) End-If

SCRHC_Step (Selective Dynamic Caching):
    If (miniCacheK > k) Then
       miniCacheK := miniCacheK - miniCacheInterval
       miniCacheIndex := miniCacheIndex - 1
    End-If
    //perform remaining hashes to obtain H^k(s)
    Code := cache[miniCacheIndex]
    For i = miniCacheK to k - 1 Do
       Code := hash(Code)
    End-If
    k := k - 1  // decrement k
    //Compute new values for the next transaction
    If (k == 0) Then Call Repeat_Chain(s, G_k, r) End-If

SCRHC_Step (Full Caching):
    Code := cache[k]
    k:= k - 1  // decrement k
    //Compute new values for the next transaction
    If (k == 0) Then Call Repeat_Chain(s, G_k, r) End-If
```

The client will send the hash value Code to the server using HTTP. When the server receives the transaction request containing the value Code from the client, it will execute the same SCRHC_Step routine using its own parameter values: s, k, r and G_k. If the Code value computed by the server matches the value received from the user, the authentication is successful.

Repeat_Chain:

The Repeat_Chain routine is an important component in the SCRHC protocol. It defines how the chain adapts and changes over time to prevent repeat-attacks and also triggers the cache values, if any, to be refreshed. In the adaptive version, the value of k is changed based on the value of r. We considered different alternatives to compute the value of k and we only present the multiplicative increase alternative in this paper. The rationale of the multiplicative increase logic is that a session that requires multiple calls to repeat the chain is likely to be a long session and the number of remaining steps in this session is likely to be relatively large. The value of k is therefore increased linearly to both accommodate short sessions and to also grow large enough to respond properly to longer sessions. Calls to the Repeat_Chain function introduce trivial computational overhead when compared to that which would be encountered from having a long chain. It is this reason that we recommend a small initial value G_k and linear growth of the chain. This multiplicative increase is intended to make the next chain loop more suitable for a long session without overshooting the value of k.

```
Repeat_Chain(s, G_k, r)
    s := hash (s || s)  //apply hash on concatenation
    r :=  r + 1
    k := Gk × r
    Call fillCache()
```

The first step in the above code changes the value of the initial secret to prevent repeat attacks. We have used the concatenation operation to change the initial secret for each new chain loop. The cache is also updated as necessary.

V. PERFORMANCE EVALUATION

To analyze the expected performance of the SCRHC protocol, we wrote a benchmark tool [1] using Java. To compare the performance of SCRHC and OTC [10], we also wrote an interpretation of the OTC solution.

It features hash chains to establish unique validation values for each transaction.

Performance comparisons were made between our protocol and OTC. Our tests modeled the limitation encountered at initialization regarding the lack of accurate knowledge of the session length (number of transactions during the lifetime of the session). For each value of the estimated session length, we averaged the results of 1000 tests in which the actual session length varied from 0.5x expected length to 1.5x expected length, randomly distributed uniformly. This was done to give an approximation as to what performance could be expected when the actual session length of a real client does not match the expected session length programmed by the website administrator. The expected session length in our tests ranged from 100 to 2500. The metrics used to evaluate performance are the number of hashes required (lower is better) and the required cache size to hold the cached hashes, if caching is used.

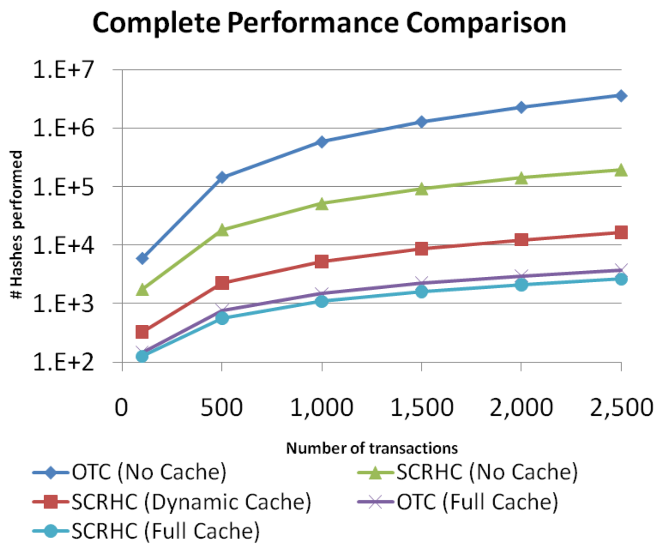

Figure 1: Complete Performance Comparison (Lower is better)

Figure 1 illustrates that all three flavors of the SCRHC protocol exceed the performance of the equivalent OTC hash chain method. For the two non-caching methods, OTC requires approximately ten times as many hashes as SCRHC. If you insist on not using caching, then our method is clearly faster, by an order of magnitude. If you are on a mobile platform and do not want to dedicate any memory to caching, then our protocol offers a significant performance advantage. Another observation from Figure 1 is the dynamic caching algorithm that we implement which yields another order-of-magnitude reduction in computation time when compared to SCRHC with no caching, while requiring minimal storage. This is ideal for those who can dedicate at least a small amount of memory for caching. Finally, you can see that the two non-caching flavors are nearly another order-of-magnitude faster than dynamic caching, but not quite. This of course comes at the expense of potentially large amounts of storage space. SCRHC still holds a 30% reduction in workload versus OTC with full caching. This leads to Figure 2, for more details.

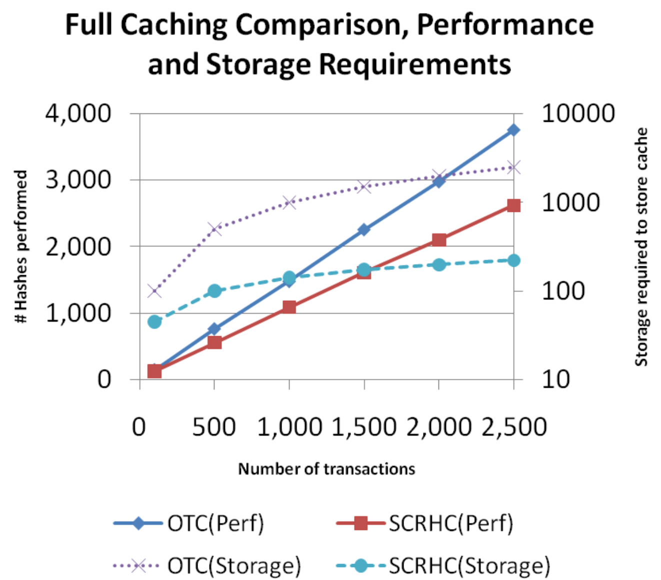

Figure 2: Comparing Performance and Storage Requirements (Lower is better)

From Figure 2 you can see that the processing requirements (solid lines, left y-axis) increase linearly with the number of expected transactions per session. SCRHC requires approximately 70% as much processing as OTC, which is admittedly not a large advantage, but the real advantage is the associated

storage requirements [in 160-bit hashes] (dashed, right y-axis). You can see that SCRHC with full caching requires around 1/10th as much storage for any expected session length. This would be very advantageous on a mobile device where storage might be limited.

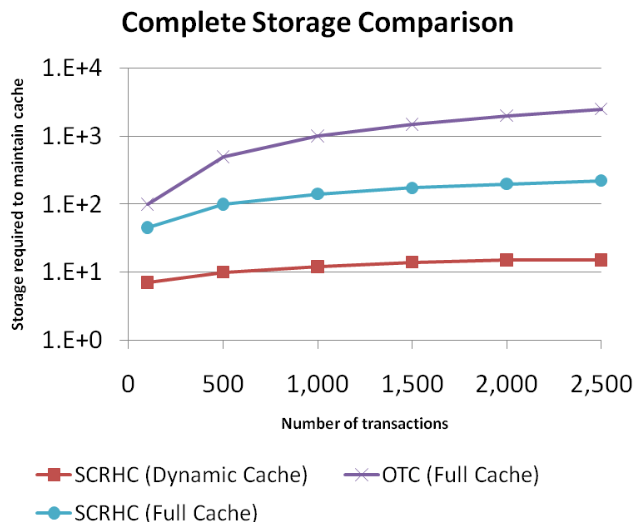

Figure 3: Comparing Requirements (Lower is better)

In Figure 3 we give a more detailed look at the storage requirements for all three flavors that require it. The important thing to note is how minimal the requirements are for dynamic caching. After analyzing Figure 1 and 3, it is clear that by shifting from OTC to SCRHC with dynamic caching, you can reduce your required processing by around 99% and only need a dozen or so hashes committed to cache.

VI. Conclusion

In this paper, we presented a novel variation of hash-chains that proved to offer better performance than similar methods while offering enough flexibility to run it on a variety of platforms. Our selective dynamic caching technique significantly reduced workloads while requiring trivial amounts of storage. We conclude that our algorithm could provide valuable security to Social Networking Sites in a flexible and adaptable manner.

References

[1] Self-Configuring Repeatable Hash-Chain Protocol Tool, http://www.jeffcashion.com/research/schrc/

[2] R. Valotta "Cookie Jacking UI redressing attacks" Presented in Swiss Cyber Storm 3 Security Conference, Rapperswil, Switzerland, May 12-15, 2011.

[3] E. Butler "FireSheep: Cookie Snatching Made Simple" ToorCon Conference, San Diego, CA, October 22-24, 2010.

[4] E. Butler and I. Gallagher "Hey Web 2.0: Start Protecting User Privacy Instead of Pretending to" ToorCon Conference, San Diego, CA, October 22-24, 2010.

[5] R. Narang "LinkedIn SSL Cookie Vulnerability" Blog posted May 21, 2011. Available at http://www.wtfuzz.com/blogs/linkedin-ssl-cookie-vulnerability/

[6] J. Cashion and M. Bassiouni "Robust and Low-Cost Solution for Preventing Sidejacking Attacks in Wireless Networks using a Rolling Code" to appear in the Proceedings of the 7th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (ACM Q2SWinet), Miami Beach, Florida, October 31-November 4, 2011.

[7] M. Huber, M. Mulazzani, E. Weippl,G. Kitzler, S. Goluch, "Friend-in-the-Middle Attacks: Exploiting Social Networking Sites for Spam," *Internet Computing, IEEE* , vol.15, no.3, pp.28-34, May-June 2011

[8] A. X. Liu, J. M. Kovacs, C. Huang, and M.G. Gouda. "A Secure Cookie Protocol." IEEE, 2005.

[9] K. Fu, E. Sit, K. Smith, and N. Feamster. "Dos and don'ts of client authentication on the web." Proceedings of the 10th USENIX Security Symposium, August 2001.

[10] I. Dacosta, S. Chakradeo, M. Ahamad, and P. Traynor. "One-Time Cookies: Preventing Session Hijacking Attacks with Disposable Credentials." Technical Report. Georgia Institute of Technology, 2011. Available at http://smartech.gatech.edu/bitstream/handle/1853/37000/GT-CS-11-04.pdf

[11] R. D. Riley, N. M. Ali, K. S. Al-Senaidi, and A. L. Al-Kuwari. "Empowering Users Against SideJacking Attacks." SIGCOMM'10, 2010.