Department of Electrical Engineering and Computer Science

Technical Report

SYR-EECS-2011-09 July 14, 2011

SCUTA: A Server-Side Access Control System for Web Applications

Xi Tan xtan@syr.edu
Wenliang Du wedu@syr.edu
Tongbo Luo toluo@syr.edu
Karthick D. Soundararaj kduraisa@syr.edu

ABSTRACT: The Web is playing a very important role in our lives, and is becoming an essential element of the computing infrastructure. Unfortunately, its importance makes it the preferred target of attacks. Web-based vulnerabilities now outnumber traditional computer security concerns. A recent study shows that over 80 percent of web sites have had at least one serious vulnerability. We believe that the Web's problems, to a large degree, are caused by the inadequacy of its underlying access control systems. To reduce the number of vulnerabilities, it is essential to provide web applications with better access control models that can adequately address the protection needs of the current Web. As a part of the efforts to develop a better access control system for the Web, we focus on the server-side access control in this paper. We introduce a new concept called subsession, based on which, we have developed a ringbased access control system (called Scuta) for web servers. Scuta provides a fine-grained and backward-compatible access control mechanism for web applications. We have implemented Scuta in PHP, and have conducted comprehensive case studies to evaluate its benefits.

KEYWORDS: Web Security, access control, web application

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

SCUTA: A Server-Side Access Control System for Web Applications

Xi Tan, Wenliang Du, Tongbo Luo, and Karthick D. Soundararaj
Department of Electrical Engineering & Computer Science
Syracuse University, Syracuse, New York, USA

ABSTRACT

The Web is playing a very important role in our lives, and is becoming an essential element of the computing infrastructure. Unfortunately, its importance makes it the preferred target of attacks. Web-based vulnerabilities now outnumber traditional computer security concerns. A recent study shows that over 80 percent of web sites have had at least one serious vulnerability. We believe that the Web's problems, to a large degree, are caused by the inadequacy of its underlying access control systems. To reduce the number of vulnerabilities, it is essential to provide web applications with better access control models that can adequately address the protection needs of the current Web.

As a part of the efforts to develop a better access control system for the Web, we focus on the server-side access control in this paper. We introduce a new concept called *subsession*, based on which, we have developed a ring-based access control system (called SCUTA) for web servers. SCUTA provides a fine-grained and backward-compatible access control mechanism for web applications. We have implemented SCUTA in PHP, and have conducted comprehensive case studies to evaluate its benefits.

1. INTRODUCTION

The Web has been growing at a rapid rate over the last 15 years. The first Google index (1998) already had 26 million pages, and by 2000 the Google index reached the one billion mark. On July 25, 2008, the one trillion milestone was reached [1]. As of May 2009, these web pages were hosted by over 109.5 million websites [2]. The Web is gradually becoming part of our lives. We do many things online, such as shopping, making friends, banking, reading news, sharing personal pictures, etc. As the most important application of the Internet infrastructure, the Web itself is becoming an essential part of the infrastructure. With such an important role the Web is playing, making sure that the Web is secure is becoming a priority for trustworthy computing.

Because of its ubiquity, the Web has become attackers' preferred target. Web-based vulnerabilities now outnumber traditional computer security concerns [3, 4]. Cross-site scripting, cross-site request forgery, and SQL injection are among the most common attacks on web applications. A recent report shows that over 80 percent of websites *have had* at least one serious vulnerability, and the average number of serious vulnerabilities per website is 16.7 [5].

It is tempting to blame developers for these security problems, because it is indeed their mistakes that have caused the problems. However, when we look deeper, asking why the percentage of vulnerabilities is so abnormally high, we soon realize that something more fundamental in the Web is wrong. One of the fundamental problems is the Web's access control system, which, being sufficient for the earlier day's Web, becomes inadequate to address the protection needs of today's Web.

The Situations. The Web, initially designed for primarily serving static contents, has now evolved into a quite dynamic system, consisting of contents and requests from multiple sources, some more trustworthy than others. Let us look at some representative scenarios. The first is *untrusted contents*. Many web applications now include user-provided contents, such as blogs, comments, and feedbacks. These are third-party data, and are less trustworthy than the first-party contents generated by the web applications themselves. If not carefully handled, malicious code can be injected into these contents.

Second, many web applications include *client-side extensions*, i.e., they include links to third-party code or directly include third-party code in their web pages. Examples of client-side extensions include advertisements, Facebook applications, iGoogle's gadgets, etc. In Figure 1, both the iPad advertisement and the weather gadget are client-side extensions. Their contents, containing JavaScript code, can be very dangerous if they are vulnerable or malicious,

Third, some web applications include *server-side extensions*, which are developed by third parties. For example, `Elgg` is an open-source social network application. It was designed as an open framework, allowing others to extend its functionality. `Elgg` already has hundreds of third-party extensions. To use these extensions, the administrators of the `Elgg` server need to install them, essentially mixing them with `Elgg`'s first-party code. These contents can also be dangerous, if they are vulnerable or malicious.

What further complicates the above scenarios is the cross-origin requests. A cross-origin request is sent from a page of one origin to a server at a different origin. Cross-origin requests are becoming quite popular nowadays. For exam-

ple, many web pages now contain small icons (e.g. Facebook icon) like those in Figure 1. Requests triggered by clicking these icons are generated by the page of `www.example.com`, but are sent to a different server. These cross-origin requests make it more convenient for users to share information among their accounts, especially their social-network accounts. Originally, cross-origin requests were only allowed for normal HTTP requests, not AJAX requests, due to obvious security concerns. Recently, however, as web technologies evolve, this restriction has been lifted to support better interactions among web applications via a protocol called Cross-Origin Resource Sharing [6]. All major browsers—Chrome, Firefox, IE and Safari—support cross-origin AJAX requests in their latest versions.

Security Needs and Problems. To secure such a complex system, good access control at the system level is essential. Without it, application developers have to include complicated protection logic in their programs to deal with the risks caused by the scenarios described above. Mistakes in the implementations of the logic, or a lack of the implementation, can cause vulnerabilities.

The Web consists of two major components, the browser and the server; access control needs to be implemented at both places. At the browser side, the access control in the current Web is based on the Same-Origin Policy (SOP), which gives the same privileges to all contents from the same origin. Such a coarse granularity, which may have been sufficient for the nascent Web in its earlier days, cannot handle untrusted contents or client-side extensions well: although these contents come from the same origin, they are not equally trusted. The inadequacy of SOP has been pointed out by various studies, and several solutions have been proposed to provide finer granularity beyond SOP [7–16], including our earlier work ESCUDO [15], which separates the contents with different levels of trustworthiness, and mediates their actions based on trust levels.

On the server side, access control is primarily based on sessions. When a user logs into a web application, the server creates a dedicated session for this user, separating him/her from the other users. Sessions are implemented using session cookies; as long as a request carries a session cookie, it will be given all the privileges associated with that session. *Namely, within each session, all requests are given the same privileges, regardless of whether they are initiated by first-party or third-party contents, from client-side or server-side extensions, or from another origin.* We would like to use an example in Figure 2 to illustrate the problem of the session-based access control.

The web page in Figure 2 allows its users to initiate three requests—`ViewFriends`, `AddFriends`, and `DeleteFriends`—from three different regions. The protection needs on the three server-side scripts are quite different: `DeleteFriends` can only be invoked by the contents that are absolutely trustworthy (e.g., the code generated by the web application itself); `AddFriends` can be invoked by the contents from semi-trusted sources, such as the code from third parties with good reputations; `ViewFriends`, due to its read-only nature, can be exposed to contents that are even less trustworthy.

Unfortunately, the current session-based access control at the web server cannot satisfy the above protection needs, neither can the existing browser-side access control solutions. Based on the current access control systems, it is very difficult to allow the three regions to access the same session, while preventing some of them from invoking certain server-side services. To achieve these two conflicting goals, applications have to implement their own ad hoc protection logic, such as asking users to confirm their actions, embedding tokens in hidden fields, etc.

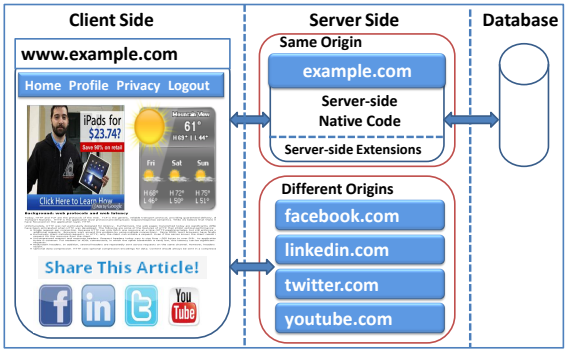


Figure 1: A web application example

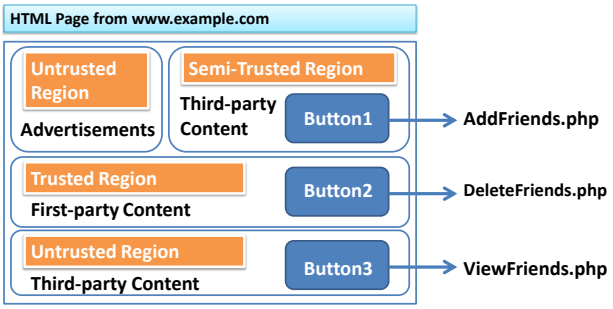


Figure 2: Diversified protection needs

The fundamental cause of the above problem is the granularity of a session: it is too coarse. The Web has become more and more complicated, and its client-side contents are no longer uniformly trusted, so requests initiated by these contents are not uniformly trusted either. Therefore, giving all the requests within the same session the same privileges cannot satisfy the protection needs of today's Web anymore. In order not to ask application developers to bear the complete responsibility of implementing those protection needs, we need a better server-side access control system.

Our Work and Contributions. The objective of our work is to develop a fine-grained server-side access control system, which can assign different privileges to the requests in the same session, based on their trustworthiness. We have developed a system called SCUTA [1], which is a novel and backward-compatible access control system for web application servers. Built upon the well-established ring model, SCUTA labels server-side data (tables in database) and programs (functions, classes, methods, or files) with rings, based on their protection needs. Programs in a lower-privileged ring cannot access data or code in a higher-privileged ring.

SCUTA divides a session into multiple *subsessions*, each

[1] "Scuta" is the plural of the Latin word "scutum", meaning a large shield used by soldiers in ancient Rome.

mapped to a different ring. Requests from a more trustworthy region in a web page belong to a more privileged subsession. Requests belonging to subsession k are only allowed to access the server-side programs and data in ring k and above (numerically). With the subsession and ring mechanisms, server-side programs can treat the requests in the same session differently, based on the trustworthiness of their initiators, and thus provide access control at a finer granularity.

We believe that SCUTA is the first system approach that intends to enhance the session-based access control system of the web server. To demonstrate its effectiveness, we have implemented SCUTA in PHP, a widely adopted platform for web applications. We have conducted comprehensive case studies to demonstrate how SCUTA can be used to satisfy the diversified protection needs in web applications.

2. BACKGROUND: ESCUDO

The server-side access control scheme described in this paper depends on the ESCUDO [15] access control on the browser, because identifying request's subsessions needs the help from browsers, especially ESCUDO-enabled browsers. We give a brief summary of how ESCUDO works in this section. We also explain why ESCUDO alone is not sufficient to deal with the security problems in web applications.

The primary objective of ESCUDO is to allow web servers to convey the trustworthiness of their contents to browsers, so browsers can use this information as the basis for the client-side access control. This provides a finer granularity than the Same-Origin Policy (SOP).

ESCUDO introduces a `ring` concept, borrowed from the Hierarchical Protection Rings (HPR) access control model [17]. Rings in ESCUDO are labeled 0, ..., N, where N is application dependent. In the HPR model, higher numbered rings have less privileges than lower numbered rings, i.e., ring 0 is the highest-privileged ring.

In ESCUDO, elements of a web page are placed into a fixed set of protection rings, based on their trustworthiness and protection needs. Elements accessible only to more trustworthy principals or from more trusted sources are placed in higher privileged rings. Ring assignments are carried out at the server side, because only the server-side code knows how trustworthy the contents are. Assigning ring labels to contents is called "configuration", and once a web page is "configured", the browser can enforce access control based on the configuration and ESCUDO's security policies.

ESCUDO uses the standard security policy in the HPR model for enforcing access restrictions inside the browser. Basically, the rule requires that the principal can only access the objects in the equal- and less-privileged rings, i.e., "no access up". It should be noted that the same origin policy is still applied here, i.e., principals from one origin cannot access the objects from another origin, regardless of which rings they are in. Rings only apply to the principals and objects from the same origin.

Rings Assignment. Browser-side contents consist primarily of two types: cookies and Document Object Model (DOM) elements. ESCUDO assigns a ring label to each cookie. Cookies that contain sensitive data should be put in a higher-privileged ring, and vice versa.

ESCUDO also assigns ring labels to DOM elements, using the HTML `<div>` tag and the `ring` attribute introduced by

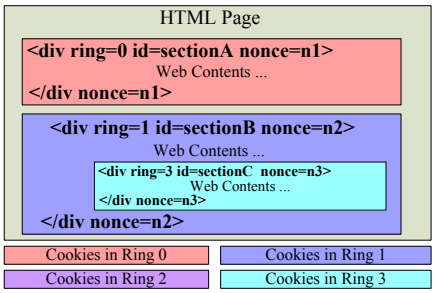


Figure 3: An Example of Escudo Configuration

ESCUDO. This `ring` attribute assigns a ring label to all the DOM elements within the scope of the tag, which is the region enclosed by the `<div>` and `</div>` pairs (Figure 3 shows an example of ring assignment). HTML allows hierarchical `div` scopes, i.e., a `div` scope can be enclosed entirely within another `div` scope. Therefore, ring assignments for DOM elements can also be hierarchical. ESCUDO ensures a *scoping rule:* the privileges of a node cannot exceed its parent's privileges, regardless of what ring label this new node has. For example, no node within `sectionB` in Figure 3 can have the ring-0 privilege: this is guaranteed by ESCUDO. Special attentions are taken to defend against the well-known *node-splitting attack* [18, 19]. The `nonce` attribute in Figure 3 is intended for that purpose.

Limitation of Escudo. ESCUDO, as well as several other client-side solutions, does a good job in mediating the access on the client side, preserving the integrity of web page, which eventually leads to a protection on the server side. However, in terms of accessing sessions, ESCUDO only provides two choices: either allowing the client-side requests to access a specific session or not allowing the access. ESCUDO achieves this by putting the session ID cookies in those rings where the session access is allowed.

The binary decision is inadequate for today's web servers. As we have shown in Figure 2, to address the protection needs in that example, at least four levels of granularity are needed: in addition to the three different needs required by the server-side scripts in the example, there is one more level, i.e., denying all. ESCUDO can only support two levels: allowing all and denying all.

Despite its limitations, ESCUDO is an indispensable component in our proposed access control framework, because it not only helps preserve the trust status from the server to the client, but also help ensure the integrity of the trust status at the browser, so no malicious contents can change their trust status. SCUTA relies on ESCUDO, and working together, they provide a more complete solution to the access control problems in web applications.

3. THE DESIGN OF SCUTA

In the current Web, the finest principal unit for server-side programs is *session*. When an HTTP request is received by a web server, the server identifies which session the request belongs to, and then gives the invoked server-side program all the privileges entitled to that session. As a consequence, all the server-side programs invoked in the same session share the same privileges. As explained earlier, this level of granularity is inadequate nowadays, be-

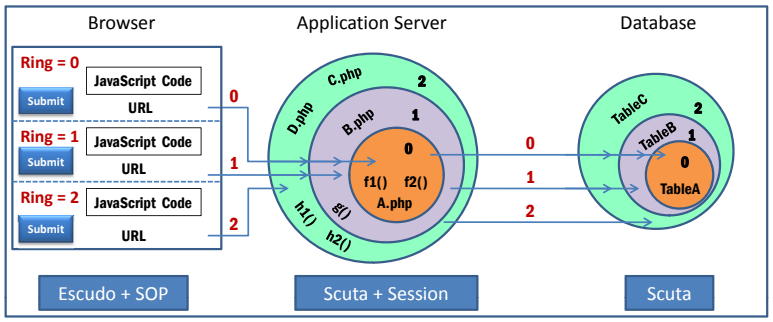


Figure 4: Server-Side Ring Mappings

cause contents in today's web applications are not uniformly trusted anymore due to the mixture of advertisements, user inputs, third-party code and active contents, etc. Actions invoked by these unequally-trusted contents should not be given the same privileges, even if they belong to the same session. This calls for a granularity level finer than session.

We have designed a novel server-side access control system called SCUTA, which provides a finer granularity than session. We describe the design of SCUTA in this section.

3.1 Subsessions

To achieve a finer granularity in access control, we divide a session into multiple *subsessions*, each identified by a subsession ID called `SubSid`. In the original web infrastructure, when a server-side program gets invoked, the server identifies which session the invocation belongs to, and then sets the corresponding running environment and protection. With the addition of `SubSid`, the server also identifies the subsession ID of the invocation.

Similar to sessions, subsessions are also implemented using cookies. In SCUTA, when a server creates a session ID, it also creates $N+1$ subsession IDs: `SubSID_0`, ..., `SubSID_N`, where $N+1$ is the total number of rings defined by the web application. These subsession IDs are sent to the browser as cookies, each marked with a different ring label: `SubSID_K` is marked with ring K. The following example shows a portion of the HTTP header generated by the server when a session is created (ESCUDO introduces a new header called `Set-Ring` to set the rings for cookies):

```
Set-Cookie: SID=pjdfbpnd228b2n; path=/
Set-Cookie: SubSID_0=gg5u1pc3inutmb
Set-Cookie: SubSID_1=h1d3vg4ep351qv
Set-Cookie: SubSID_2=n91n6kgiv05fe2
Set-Ring: SID=2
Set-Ring: SubSID_0=0
Set-Ring: SubSID_1=1
Set-Ring: SubSID_2=2
```

With ESCUDO's access control rules, contents in ring t can access the subsession cookies in rings t and above (numerically). Therefore, when an HTTP request is made in ring t from an ESCUDO-protected webpage, all the subsession cookies from rings t and above will be attached to the request; the server can use these cookies to decide that the request belongs to subsession t. Figure 5 depicts the subsession-identification process. In the figure, the server-side program `F.php` is invoked under two different scenarios: one by the contents (button, JavaScript code, or link) in ring 0, and the other in ring 2. With the help of ESCUDO, SCUTA can successfully identify the subsession IDs of these invocations.

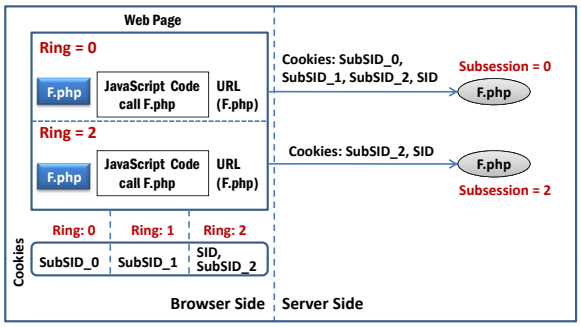


Figure 5: The Subsession mechanism

It should be noted that the session ID (SID) is placed in the lowest-privileged ring (ring 2 in this example), allowing it to be carried by all requests, so servers can identify what session a request belongs to.

3.2 The SCUTA Ring Model

Using subsessions, servers can clearly tell from which ring a request is made, and thus know how trustworthy the request is. This is essential for access control, but we need something more: we need to allow web applications to assign privileges to each subsession. The ring model, used by ESCUDO to configure web pages, can be naturally applied to configure application servers. Figure 4 depicts an overall picture of this model in SCUTA.

To use ESCUDO for access control at the browser side, web applications need to generate a set of rings, and configure web pages using these rings. SCUTA uses the same set of rings, but instead of configuring web pages, it configures the server-side programs. Namely, SCUTA places server-side programs into different rings, based on their protection needs. The trustworthiness given to a particular ring in servers is the same as that in browsers. To avoid confusion, we use "ESCUDO ring" and "SCUTA ring" to refer to the ring configuration in web pages and at servers, respectively.

Once the server-side programs are labeled with SCUTA rings and the subsessions of HTTP requests are identified, SCUTA can conduct fine-grained access control on those requests. We have the following basic access rule:

DEFINITION 3.1. **Basic Access Rule:** *An HTTP request originating from* ESCUDO *ring t on the client side will be identified as belonging to subsession t; it is allowed to invoke the server-side programs in* SCUTA *ring w if $w \geq t$.*

The above rule does not only apply to the entry programs at the server side (e.g. `DeleteFriends.php` in Figure 2), it also applies to all the functions invoked during the execution of the programs. These functions include stand-alone functions, methods in classes, as well as scripts "invoked" via the `include` command in PHP. These functions are labeled with the ring information in a configuration file (see the implementation details in Section 4).

Using the ring-based access control in SCUTA, web applications can put their security-critical server-side programs in higher-privileged SCUTA rings, limiting their accesses only to the web-page contents in the corresponding ESCUDO rings. Since the less trustworthy web-page contents are put in less-privileged ESCUDO rings, even if they, due to security breaches, contain malicious code, forms, or URL links, they are prevented from accessing the security-critical programs placed in the higher-privileged SCUTA rings. For example, the security-critical `DeleteFriends` program in Figure 2 can be put in a high privileged SCUTA ring, such as ring 0.

Some functionalities of web applications are less security-critical, and they can be put into lower-privileged rings to allow broader access. For example, `ViewFriends` does not involve modification, so it is less security sensitive; if the web application wants to allow `ViewFriends` to be accessible by third-party JavaScript code, or even by requests from another origin (i.e. cross-origin request), they can simply put this program in the less-privileged SCUTA rings, allowing less-trustworthy client-side contents to invoke it.

Database Protection using Rings. SCUTA's ring-based access control also applies to databases. Namely, data in databases are also mapped to rings, such that a request in subsession t can only access the data in rings t and above. In particular, we label tables of databases with rings depending on the level of protection required by web applications. For example, in a social-network application, the profile table is usually security sensitive, so its access should be restricted to ring 0. To achieve that, we put the profile table in ring 0. For tables that are less security sensitive, we put them in the lower-privileged rings, allowing a broader access. The right side of Figure 4 depicts the ring configuration of databases.

SCUTA's ring configuration is not necessarily limited to the table level; it supports a much finer granularity. For example, SCUTA can label a table in a way, such that it is read-only in ring 3, but writable in ring 0; SCUTA can also label some columns of a table with one ring, while labeling other columns with another ring. Such a fine granularity provides a great flexibility to web applications. More sophisticated examples are given in Appendix B.

3.3 Cross-Ring Invocations

In real-world applications, cross-ring invocations are often desirable. There are two types: from lower to higher privileged rings and vice versa. We discuss how SCUTA supports these types of invocations.

From lower to higher privileged ring. As we have learned from many other ring-based access control systems, such as those in operating systems and 80x86 CPUs, disallowing invocations from lower to higher privileged rings tends to be over restrictive; in many cases, such an invocation needs to be supported. For example, in operating systems, user-level programs are basically running in a less privileged ring, but to access files, they need to invoke the code in the kernel (running in a higher privileged ring). That is a cross-ring execution. Operating systems support that with system calls, the essence of which is to support a controlled invocation from lower to higher privileged rings.

To support the similar kind of invocation, borrowing from operating systems, we introduce the *gate* concept in SCUTA. Its definition and rules are described in the following:

DEFINITION 3.2. **Gate Access Rule:** *A gate is a function labeled with the* `GATE` *keyword and a tuple* (R, W)*, where* R *is the ring that this function belongs to, and* W *is the threshold, representing the highest ring number (i.e. the least privileged ring) that is allowed to access this gate. Subsession* t *can invoke a gate function with the label* (r, w) *if* $r \leq t \leq w$*, i.e., the subsession is less privileged than ring* r*, but compared to the threshold* w*,* t*'s privilege is sufficient.*

With gates, web developers can write "system calls" for their applications, providing a "gate" for the less privileged code to invoke more privileged code, in a controlled fashion. For example, as in Figure 2, we may want to allow some third-party extensions in ESCUDO ring 1 to invoke the `DeleteFriends` function, as long as the user specifically grants the permission (i.e., the function will provides extra application-specific control to safe-guard the friend list). However, because of protection needs, the `Friend` table is stored in SCUTA ring 0, forcing the `DeleteFriends` function to be put in ring 0 as well, and denying invocation by the third-party JavaScript code (in ESCUDO ring 1). This dilemma can be solved using gates: we can keep the `DeleteFriends` function in ring 0, but label it as a gate, and assign $(0, 1)$ to it. Such a configuration allow code in rings 0 and 1 to invoke `DeleteFriends`, which will be executed in the context of SCUTA ring 0.

From higher to lower privileged ring. SCUTA allows code in higher-privileged rings to invoke that in lower-privileged rings. This is essential; for example, in Figure 2's example, the function `AddFriends` is in placed in SCUTA ring 1, but we do want the contents in ESCUDO ring 0 to invoke it. Obviously, allowing such an invocation brings risks, especially when the callees are not trustworthy. Putting a function in less privileged rings does not necessarily mean this function is less trustworthy (e.g. shared libraries are often put in the least privileged ring), but a less trustworthy function must be put in less privileged rings.

Many web applications also allows servers to import third-party code. For example, ELGG, a popular social-network application, provides an open framework to developers, who can develop interesting applications that can be added to ELGG. These third-party ELGG extensions may be buggy, or even malicious potentially. With SCUTA, they can be put in a less privileged ring. However, if the client-side contents in subsession 0 or the server-side code in ring 0 invokes these extensions, the execution will take the caller's privilege (i.e., ring 0). This becomes dangerous. Therefore, we have the following access rule:

DEFINITION 3.3. **Privilege Downgrading Rule.** *When a caller running in ring* t *invokes a function in a less privileged ring* $w > t$*, the effective subsession ID during the execution of this function is downgraded to* w*; the caller regains its effective subsession ID* t *after the function returns.*

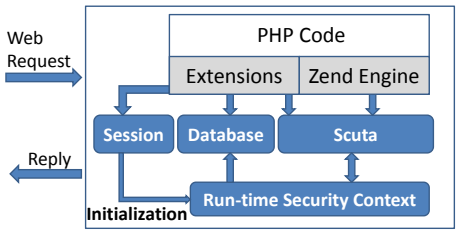


Figure 6: The Architecture of the Scuta System

3.4 Supporting Other Access Control Models

Discretionary Access Control. All access control models have their limitations, especially for simple models that are intended to address the generic needs; there will be security needs that cannot be covered. For instance, in operating systems, some applications, such as Unix's SetUID programs, need to enforce their discretionary security control, which cannot be easily addressed by the generic model provided by operating systems. The enforcement is thus implemented in the program logic. Therefore, a good access control model should provide primitives to enable the implementation of discretionary access control by applications.

SCUTA provides a primitive called `session_esubsid()`, which returns the effective subsession ID of the current execution. With this API, programs can enforce their application-specific security policies based on subsession IDs. For example, if a program wants to perform different tasks for different subsessions, it can do the following:

```
switch (session_esubsid())  {
  case 0: Do task A; break;
  case 1: Do task B; break;
  case 2: Do task C; break; }
```

Other Access Control Models. SCUTA chooses to enforce the multi-level ring model, because it intends to conduct access control based on trust, and trust, by natural, has multiple levels. As the Web evolves, needs for other models, such as multilateral models, may arise. We do believe that the subsession concept introduced by SCUTA can be adapted to serve as the basis for those models; obviously, the access control rules need to be redesigned if model changes. We will study other models in our future work.

4. IMPLEMENTATION OF SCUTA

4.1 The Architecture of the SCUTA System

SCUTA is a general model that can be implemented in various platforms, including PHP, Java Servlet, and ASP.NET. In this work, we choose the open-source PHP platform in our implementation. In order to implement SCUTA, we need to change the behavior of PHP to enforce the ring-based access control policy during the execution of PHP code. Normally, this requires a modification of the target system. However, the PHP architecture was designed for extensibility, allowing developers to add new or modify the existing functionalities.

The PHP architecture has two major components: core and extensions. The core focuses on setting up the running environment, file streams, error handling, etc, which are essential functionalities of PHP. Besides, the core provides an interface for loading additional functionalities (called *extensions*). It is these flexible extensions that make PHP one of the most popular choices for web applications. Most PHP functionalities familiar to developers are actually extensions. For example, the session and database-access mechanisms are all implemented as extensions.

At the center of the core lies a virtual machine, called the Zend Engine, which parses PHP scripts into opcodes, and then executes them. The Zend Engine was also designed for extensibility. It not only allows the overriding of its basic functionality, including compilation, execution, and error handling, but also allows developers to add new functionalities through a set of well-defined hooks [20]. Appendix A provides more details about these hooks.

PHP's extensible architecture makes it quite convenient to implement SCUTA in PHP. The implementation of SCUTA involves three PHP extensions: session, database, and Scuta. The first two are existing extensions that need to be modified, and the third one, SCUTA's access control engine, is a new extension created by us. These three extensions depend on a common security context, which serves as the basis for our access control. The main element of this security context is the *effective subsession ID*, which indicates the effective ring of the current execution. The session module initializes this security context, the database module uses the context, while the Scuta module uses and updates this context. A high-level overview of the SCUTA system is depicted in Figure 6, and we will discuss these modules in details.

4.2 The SCUTA Access Control Module

The main access control engine of SCUTA is implemented as a PHP extension (called Scuta). Its primary goal is to enforce SCUTA's ring-based access control during the execution. SCUTA's access control is conducted at the function level, i.e., when a function is invoked, SCUTA needs to decide whether the invocation is allowed or not. To achieve this, we need to intercept function calls during the runtime.

The Scuta extension intercepts function calls using the hooks provided by the Zend Engine. As we mentioned before, the Zend Engine provides a number of hooks, allowing extensions to insert additional code at particular places. In our implementation, we mainly used two hooks: one on function entry, and the other on function exit. At the function entry point, the Scuta extension checks whether a function can be invoked; if not, it throws a fatal error, causing the program to terminate. If the invocation is allowed, the Scuta extension updates the security context, and then gives the control to the invoked function. When the function returns, the Scuta extension takes control again via the function-exit hook, and updates the security context.

The Ring Configuration File. We use a configuration file to map program directories, files, classes, class methods, and functions to rings. This configuration is loaded into a hash-table when the web server is started. During the runtime, for each function (or method), SCUTA can get all its information, including function name, class name (for methods only), file name, and directory name. SCUTA then searches for the ring information of this function from the hash-table in the following order: (1) use the function name or method name, (2) use the class name (only for methods), (3) use the file name, (4) use the directory name, and (5) if all fail, set the subsession to the least privileged ring (i.e., SCUTA's default setting).

Labeling server-side programs is done outside of the pro-

grams; therefore, setting the security policy in SCUTA is separated from the program logic, and achieved using "configuration", instead of "implementation".

4.3 The Session Module

In the web applications that use PHP's built-in session mechanism, the function `session_start()` needs to be called at the beginning of a program. If the session does not exist, i.e., the HTTP request does not contain a Session ID (SID) cookie, this function will generate an SID cookie, and set the cookie in the header of the reply. When the client gets the reply, it will store the SID cookie in the browser, and attach the cookie to the subsequent HTTP requests bound to the same server. When serving subsequent HTTP requests, `session_start()` will still be invoked, but now seeing the SID cookie, it will not create a new session; instead, it will resume the existing session identified by the SID, as well as loading the session data.

To implement the subsession mechanism, we modified the `session_start()` function in the session extension. We added two functionalities. First, when a new session is created, subsessions will be generated, and the subsession cookies will be sent to the browser, along with the session cookie. Details of this process are already given in Section 3.1. Second, when an HTTP request comes, carrying subsession IDs, `session_start()` identifies the request's subsession ID based on the subsession cookies carried by the request. Then the function initializes the runtime security context. Both functionalities are quite easy to implement. The modification only involves about 120 lines of code.

Backward compatibility issue. In our implementation, instead of using the standard `Set-Cookie` to set the subsession cookies, we decided to define a new header called `Set-CookieSub` for that purpose (only for setting the subsession cookies; the session cookie is still set using `Set-Cookie`). This is mainly for backward compatibility. In an ESCUDO-enabled browser, `Set-CookieSub` is equivalent to `Set-Cookie`. However, in a non-ESCUDO browser, the `Set-CookieSub` header will be ignored, so no subsession ID will be set as cookies on the browser side; therefore, requests from a non-ESCUDO browser will not attach any subsession ID. In this way, servers can tell whether a browser is ESCUDO-enabled or not. If not, the server will automatically assign the lowest-privileged subsession ID to the requests, and thus providing the minimal services to non-ESCUDO browsers.

4.4 The Database Modules

PHP-based web applications interact with databases using the APIs provided by several PHP extensions, such as `mysql` and `mysqli` for MySQL databases. Before PHP programs send a query to databases, they use these APIs to establish a connection with the database. For example, `mysql_connect` in the `mysql` extension and `mysqli_connect` in the `mysqli` extension are the APIs for this purpose. These APIs require a user name and a password; in most web applications, PHP connects to databases through a single user account. We use `dbuser` to refer to this account in our discussions.

To enforce the ring-based access control in databases, we leverage the databases' built-in access control mechanism, which can grant different database-access privileges to different users. Our basic idea is to create several new user accounts, one for each ring, so we can use the database's user-based access control. In particular, in our MySQL implementation, for the `dbuser` account, we create accounts `dbuser_0`, `dbuser_1`, and `dbuser_2` (assuming there are only three rings), with `dbuser_t` corresponding to ring t. We then use MySQL's `GRANT` command to grant each `dbuser_t` the database-access privileges entitled to ring t. This is achieved by the database administrator from inside the database. For example, if we want to put `TableA` in ring 0, `TableB` in ring 1, and `TableC` in ring 2, we run the following `GRANT` commands:

```
GRANT ALL ON TableA TO dbuser_0;
GRANT ALL ON TableB TO dbuser_0;
GRANT ALL ON TableC TO dbuser_0;
GRANT ALL ON TableB TO dbuser_1;
GRANT ALL ON TableC TO dbuser_1;
GRANT ALL ON TableC TO dbuser_2;
```

We modified `mysql_connect()`, so when a program wants to connect to the database using the `dbuser` account, we replace this account name with `dbuser_t`, where t is the effective subsession ID. Therefore, subsequent queries using this connection are bounded by the privileges assigned to `dbuser_t`, i.e., they can only access the tables in rings t and above; Similar changes are made in `mysql_pconnect()`. Moreover, many applications use `mysqli_connect()` of the `mysqli` extension to connect to the database. We thus made corresponding changes to `mysqli`. The total changes are less than 30 lines of code.

MySQL's `GRANT` command allows us to conduct access control at even finer levels, including table columns, type of database operations, etc. For example, we can place a table in both ring 0 and ring 2, but ring 2 can only read this table and update its column k, while ring 0 can update the entire table. More details are given in Appendix B.

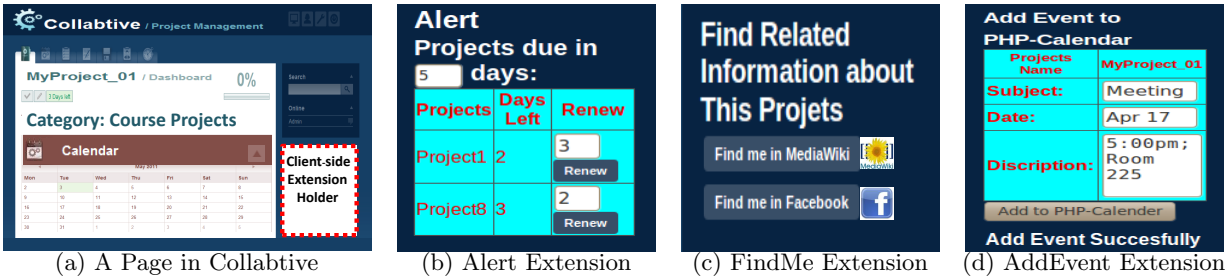
Configuration tool. To avoid mistakes caused by manually running the `GRANT` command, we created a configuration tool, allowing developers to specify the ring configuration in a file. The file will be loaded by MySQL when it starts, and the configuration tool will then turn the ring configuration into corresponding `GRANT` commands. The format of the configuration file is given in Appendix B.

5. CASE STUDIES AND EVALUATION

To evaluate how SCUTA helps secure web applications, we have conducted five case studies using several open-source web applications, including `Collabtive` (a web-based project management system), `Mediawiki` (a wiki system), and `PHP-Calendar` (a web calendar). We use `Collabtive` as the basis (Figure 7(a)), which has a holder for client-side extensions. We have developed three different extensions for the demonstration purpose: Alert (Figure 7(b)), FindMe (Figures 7(c)), and AddEvent (Figure 7(d)). The case studies are divided into two major categories: protecting same-origin requests and protecting cross-origin requests.

5.1 Protecting Same-Origin Requests

Case 1: Client-side extensions. Most web applications nowadays include client-side extensions developed by third parties. These extensions, usually including JavaScript code, are embedded in web pages and executed at the browser side. To demonstrate the benefit of SCUTA, we created a client-side extension called Alert for `Collabtive` (Figure 7(b)). This extension primarily displays a user's upcoming project



(a) A Page in Collabtive (b) Alert Extension (c) FindMe Extension (d) AddEvent Extension

Figure 7: Collabtive and Its Client-Side Extensions

deadlines; session cookies need to be attached to its requests to get the user's project information. In the current Web, that means the extension is granted all the user's privileges and can do a lot of damages to the user. Extensions like this have become quite popular in the Web. Client-side extensions are often developed by third parties, so they are less trustworthy than the web application's first-party contents. If they are malicious or vulnerable, the entire web application becomes endangered.

Because of the needs for client-side extensions to execute JavaScript and access session cookies, most of the existing methods, such as iframe, character escaping, NoScript add-ons [21] and ESCUDO, cannot easily achieve the desirable protection when including client-side extensions. In fact, all these methods either disallow JavaScript or disallow the access to session cookies; none supports the Alert extension. Without an appropriate protection mechanism, most web sites have to resolve to code verification and examination, filtering out malicious and vulnerable extensions, a practice that is complicated and error-prone.

SCUTA provides an intuitive and systematic mechanism to protect web applications against malicious/vulnerable client-side extensions. Using SCUTA, we can place client-side extensions, if not fully trusted, in a lower privileged ESCUDO ring within web pages, and accordingly, place its required server-side functions in the same SCUTA ring. Meanwhile, we place all the sensitive server-side functions, such as delete, update and insert, in the higher privileged SCUTA rings. Therefore, the not-fully-trusted client-side extensions will be limited to access the less sensitive server-side programs, such as displaying and viewing functions. The damage is greatly limited if they are compromised.

In our demonstration in Figure 7(b), we put the Alert client-side extension in ESCUDO ring 3. To allow it to get the project deadline information, we place the read-only server-side function `display()` in SCUTA ring 3, while placing all the modification-involving functions in SCUTA ring 0. Therefore, if this extension is malicious, it can steal information, but cannot modify anything on the server side.

Case 2: Gate. Our Alert client-side extension in the previous case also has a `renew` button, which allows the user to postpone a project's deadline by at most a week. The extension needs to invoke some server-side function to modify project deadlines. The PHP function `add()` allows such a modification, but in addition to deadlines, it can also be used to modify many other aspects of projects (e.g., project titles). To prevent it from being invoked by untrusted client-side contents, this function is placed in SCUTA ring 0, essentially preventing the access by our Alert client-side extension. To solve this dilemma, we use Gate in SCUTA.

Gate is like system calls in operating systems, allowing lower-privileged code to invoke higher-privileged code, but in a controlled fashion. We created a Gate function called `renew()`, which calls `add()` to postpone project deadlines only, but before making the call, it verifies that the requested postpone duration is within a week. We place the Gate function `renew()` in SCUTA ring 0, but allowing accesses from subsession 3 (i.e. the threshold value in the gate specification is set to 3). Therefore, the Alert client-side extension, placed in ESCUDO ring 3, can invoke `renew()`, which runs in SCUTA ring 0 and can thus call `add()` to modify deadlines. Essentially, `renew()` provides a controlled access to the security-sensitive function `add()`.

Case 3: Server-side extensions. Server-side extensions of web applications are server-side programs developed by third parties. Examples of server-side extensions include `Elgg`'s social-network applications, third-party libraries, etc. These extensions are intended to extend the functionalities of web applications, and they are installed at the server side by web applications' administrators.

The current Web's security infrastructure is unable to deal with the risk introduced by server-side extensions, because these extensions, once installed, have the same privileges as the native code of web applications: they can directly manipulate databases, invoke the security-critical APIs, etc. This situation is dangerous if the extensions are vulnerable, or even worse, malicious.

In Figure 7(a), the functionality to display the category information was actually added by us. It sends an AJAX request to the server-side program `DisplayCat`, which fetches and returns the category information from the database. `DisplayCat` was also implemented by us (i.e., third-party developers), and was incorporated in `Collabtive` as a server-side extension. Because of our oversight, there was a SQL injection vulnerability in the code[2]. That is, a user, legitimate one, can intercept the AJAX request (e.g. using Firefox Add-ons), modify the value of its parameters using the SQL injection technique, and can thus cause `DisplayCat` to execute a malicious SQL statement. The situation will be more devastating if `DisplayCat` is malicious, because once triggered, it can run whatever PHP and SQL code it wants, essentially attacking `Collabtive` from the inside.

With SCUTA, we can reduce the risks introduced by server-side extensions. This is done by placing the untrusted server-

[2]SQL injection vulnerability is one of the most common vulnerabilities in web applications [22].

side extensions to less privileged rings (say ring K). When client-side requests or other server-side programs invoke a function in this extension, the function will be executed only with the privilege of ring K. In the example in Figure 7(a), the `DisplayCat` function is placed in ring 3. Since no database-update privileges or security-sensitive PHP functions are placed in ring 3, even if `DisplayCat` is vulnerable or malicious, and even if it is invoked from subsession 0, according to SCUTA's *privilege-downgrading rule*, it will be executed in the context of subsession 3; its damage is thus limited.

5.2 Protecting Cross-Origin Requests

A cross-origin (or cross-site) request is sent from a page of one origin to a server at a different origin. Cross-origin requests are becoming quite popular nowadays. We study how SCUTA can help secure these requests. There are two types of cross-origin requests: non-AJAX and AJAX.

Case 4: Cross-origin non-AJAX request. The non-AJAX type is typical cross-site HTTP requests. A good example is the Facebook button included in many web pages (see examples in Figure 1 and Figure 7(c)). When these requests are made, browsers attach all the cookies of the targeted site to the requests. On one hand, cross-origin requests are widely used by web applications, but on the other hand, this type of requests are the culprit of the cross-site request forgery attack (CSRF).

To allow cross-origin requests while protecting against the CSRF attack, developers have to implement specific protection logic in their applications. A common practice is to embed secret tokens in web pages, so they can only be attached to the requests from these pages (i.e., same-origin requests), not the cross-origin requests. At the sever side, web applications add extra program logic to check the existence of the secret tokens. If developers miss a place, CSRF may be possible. `Collabtive` uses the secret-token approach, but unfortunately, it only places the checks in 6 out of the 16 server-side programs that need to be protected against the CSRF attack. `MediaWiki` has a similar situation, having limited CSRF protection.

ESCUDO is also quite limited in protecting cross-origin requests. The way how ESCUDO defends against CSRF attacks is to only allow the cross-origin requests to access the target-domain's cookies in the lowest-privileged ring. Because the session cookies are placed in ring 0 by servers, the cross-origin requests cannot access the session cookies, and thus unable to launch CSRF attacks. Unfortunately, this defense breaks many useful applications, such as the Facebook button in Figure 7(c), because they do need session cookies, without which, every time a cross-origin request is sent, the user must be authenticated repeatedly—very inconvenient.

SCUTA provides a better approach to protect cross-origin requests. It allows cross-origin requests to use sessions without becoming a victim of the CSRF attack. The basic idea is to only expose those CSRF-safe services to the cross-origin requests. CSRF-safe means that even if a request is forged, it can do no harm to the server. For example, the services that only provide information to users without modifying anything on the server are CSRF-safe. Using SCUTA, developers can place those CSRF-safe services in the least-privileged ring. This ring is accessible to the cross-origin requests, because ESCUDO maps the cross-origin requests to the least-privileged ring of the target domain (and thus the request will be treated as from the least-privileged subsession). For those CSRF-unsafe services, developers must place them in the more privileged rings, essentially denying the access from cross-origin requests. If controlled accesses to these services are needed, we can use gates.

Compared with the existing CSRF countermeasures, the protection achieved by SCUTA is simple and systematic, and needs only configuration, not implementation. The existing CSRF countermeasures need to be placed in the program logic and repeated for every CSRF-unsafe functions. We believe that a well-designed access control system like SCUTA can simplify application developers' tasks by enforcing much of the access control within the model, freeing developers from such a complicated and error-prone task.

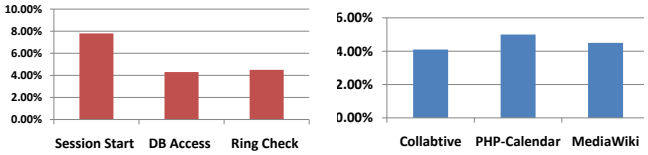
Case 5: Cross-origin Ajax request. Due to security risks[3], making cross-origin requests using AJAX was initially disallowed by browsers based on the same-origin policy. Recently, however, to allow better interactions among web applications, this restriction has been lifted by almost all major browsers. A protocol called Cross-Origin Resource Sharing (CORS) [6] was introduced to support such type of requests. CORS uses HTTP's newly introduced `Origin` header to identify the origin of cross-origin requests; application servers can decide whether to allow the access from the specified origin.

We made an example in Figure 7(d), in which, users of `Collabtive` can add project-related events to their calendars at `PHP-Calendar`. The request, from `Collabtive` pages to the `PHP-Calendar` server, is a cross-origin AJAX request. To allow such a request, we need to add an origin-checking logic in the `PHP-Calendar`'s `AddEvent` service, checking whether a cross-origin request is from `Collabtive` or not; if not, requests will be denied. Essentially, `PHP-Calendar` puts `Collabtive` on its trusted white-list for the `AddEvent` service.

Such a trust is too coarse-grained and risky: when putting `Collabtive` on its white-list, `PHP-Calendar` automatically delegates the trust to all the contents in the `Collabtive`'s pages, regardless of whether they are `Collabtive`'s first-party or third-party contents. If those third-party contents are vulnerable or malicious, attackers can take advantage of the trust, and launch attacks on `PHP-Calendar` from `Collabtive`. Therefore, if cross-origin AJAX requests are initiated from these third-party contents, `PHP-Calendar` should not treat them the same as those from `Collabtive`'s first-party contents. This distinction is not feasible in the current Web system.

SCUTA's subsession mechanism can achieve such a distinction, enabling web applications to conduct access control on cross-origin AJAX requests at a finer granularity than the current practice. In our case study, the EventAdd gadget is placed in ESCUDO ring 1. When it makes a cross-origin AJAX request to `PHP-Calendar`, it will be recognized by `PHP-Calendar` as belonging to subsession 1. The `AddEvent` function is placed in SCUTA ring 1 at the `PHP-Calendar` server, allowing the cross-origin access from the `Collabtive`'s EventAdd gadget. If some untrusted contents on the same page try to access `AddEvent`, as long as `Collabtive`

[3]Replies to the cross-origin non-AJAX requests are handled by browsers, which are trustworthy; for the AJAX requests, the replies are handled by the JavaScript code that sent the cross-origin requests; the code may not be trustworthy.



(a) Individual Overhead (b) Overall Overhead

Figure 8: Performance Evaluation

puts these contents anywhere above ESCUDO ring 1, the access will be denied by SCUTA.

If `PHP-Calendar` trusts `Collabtive` less, it can map `Collabtive`'s requests from ESCUDO ring K to subsession K+Δ, essentially downgrading the privileges given to the cross-origin requests from `Collabtive` by Δ rings.

5.3 Performance Evaluation

To evaluate the performance of SCUTA, we measured the server-side execution overhead caused by SCUTA. We used an Apache benchmarking tool called `ab` in our experiment. The tool allows us to measure the processing time of each HTTP request. Network latency is already excluded in our measurement, as it has nothing to do with SCUTA.

We have designed two sets of experiments: The first set focuses on evaluating the overhead of individual operations that are affected by SCUTA. These operations involve session/subsession initialization, database access, and SCUTA ring check. For each operation, we created a small program that only consists of that operation, we measured the execution time, with and without SCUTA, and we calculated the overhead of SCUTA. The results are plotted in Figure 8(a), which indicates that the overhead on individual operations is quite small, especially for database access and SCUTA ring check (only around 4%); session-start needs to generate and verify subsession IDs, and the extra encryption/decryption causes more overhead (around 8%).

In the second set of experiments, we measured the overall execution overhead that SCUTA brings to real web applications. We used `Collabtive`, `MediaWiki` and `PHP-Calendar`, and plotted the results in Figure 8(b). The results show that the overall overhead caused by SCUTA is quite small, only about 4%.

6. RELATED WORK

The need for providing a fine-grained access control on the server-side of the web infrastructure has been well recognized by many researchers. A number of language-based approaches have been proposed in the past. Early language-based work focuses on enforcing confidentiality and integrity of web applications. They use static analysis methods to help enforce fine-grained security policies [23, 24], or use dynamic tainting methods to detect vulnerabilities [25].

Recent language-based work starts to focus on enforcing fine-grained access control at the framework level. Chong et al. proposed a novel framework, called SIF [26], to build web applications. With SIF, explicit confidentiality and integrity policies can be given as a compile-time program annotation or as run-time user requirement. Compile-time and run-time checking will enforce these policies. Another language-based approach is called Capsules [27] by Krishnamurthy et al. This framework benefits from an object-capability language called Joe-E [28]. Capsules provides interfaces that expose limited, explicitly-specified privileges to application components. With Capsules, the web framework can enforce privilege separation and isolation of web applications, and thus restricts what each component of the application can do and quarantines buggy or compromised code.

The goal of SCUTA and the language-based approaches is the same, i.e., to achieve a fine-grained access control in web applications, but SCUTA takes a very different approach. SCUTA provides a new subsession primitive for web applications, so requests from trusted client-side contents can be separated from those from not-so-trusted contents; such a separation enables web applications to enforce a fine-grained access control. This kind of separation cannot be achieved by the existing approaches. Actually, SCUTA and the existing language-based approaches complement each other quite nicely. For instance, web developers can use SCUTA to map the programs to different subsessions, and then use the language approaches to further restrict the programs within each particular ring; the subsession concept can also be integrated into the security policies that are enforced by the language approaches.

There are numerous studies that focus on enforcing fine-grained access control at the client side, including Caja [10, 11], ConScript [12], Content Security Policy [29], work by Maffeis at al. [30], etc. This body of work focuses on the client-side code, while SCUTA focuses on the server-side code. SCUTA does use ESCUDO [15], which is one type of client-side protection scheme. In our future work, we will explore how SCUTA can take advantage of the client-side protections achieved by the other work.

There is also a large body of work in defending against specific attacks on web applications, including cross-site scripting attacks, cross-site request forgery attacks, SQL-injection attacks, etc. Summaries of this body of work can be found in [12, 15, 29, 30]. SCUTA does not focus on any specific attack; it focuses on providing a better access control. As a natural result of this access control, web applications can be better protected against these attacks.

In addition to the session-based architecture currently used by most web applications, researchers are also exploring other alternatives [31, 32]. SCUTA may not be directly applicable to those new architectures, but we believe that ideas similar to SCUTA may be used to provide a fine-grained access control in their security infrastructures.

7. CONCLUSION

To improve the security in web applications, we have designed SCUTA, a fine-grained access control system for web servers. SCUTA is based on the subsession concept that we introduced to the web infrastructure. Subsessions allow servers to distinguish more trustworthy HTTP requests from less trustworthy ones. Such a distinction can help servers enforce fine-grained access control. We have developed a ring-based access control model based on subsessions. We have implemented SCUTA in PHP. Using case studies, we have demonstrated that SCUTA can be used by web applications to satisfy a variety of protection needs, most of which are hard to satisfy using the Web's current access control systems. In our future work, we plan to further develop configuration and automation tools to help users configure their web applications, and detect problems in their configuration.

8. REFERENCES

[1] J. Alpert and N. Jesse, "We knew the web was big..." The Official Google Blog. http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html, 2008.

[2] N. Intelligence, "Domain counts & internet statistics," http://www.domaintools.com/internet-statistics/, May 2009.

[3] S. Christey and R. A. Martin, "Vulnerability type distributions in cve (version 1.1)," MITRE Corporation. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007.

[4] S. Corp., "Symantec internet security threat report: Trends for july-december 2007 (executive summary)," Page 1–2, 2008.

[5] WhiteHat Security, "Whitehat website security statistic report, 10th edition," 2010.

[6] "Cross-origin resource sharing," URL: http://www.w3.org/TR/cors/, 2010.

[7] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting browser state from web privacy attacks," in *WWW 2006*.

[8] B. Livshits and U. Erlingsson, "Using web application construction frameworks to protect against code injection attacks," in *PLAS 2007*.

[9] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *CCS 2007*.

[10] "Caja," http://code.google.com/p/google-caja/.

[11] D. Crockford, "ADSafe," http://www.adsafe.org.

[12] L. A. Meyerovich and V. B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser," in *IEEE Symposium on Security and Privacy*, 2010, pp. 481–496.

[13] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig, "CLAMP: Practical prevention of large-scale data leaks," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[14] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: Preventing authentication & access control vulnerabilities in web applications," in *Proceedings of the Eighteenth Usenix Security Symposium (Usenix Security)*, Montreal, Canada, 2009.

[15] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin, "Escudo: A fine-grained protection model for web browsers," in *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*, Genoa, Italy, June 21-25 2010.

[16] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in javascript contexts," in *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, Minneapolis, Minnesota, USA, June 20-24 2011.

[17] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," *Commun. ACM*, vol. 15, no. 3, pp. 157–170, 1972.

[18] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *WWW 2007*.

[19] M. V. Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2009.

[20] G. Schlossnagle, *Advanced PHP Programming*. Sams, 2004.

[21] "Noscript add-ons," URL: https://addons.mozilla.org/en-US/firefox/addon/noscript/.

[22] OWASP, "The ten most critical web application security risks," http://www.owasp.org/index.php/File:OWASP_T10_-_2010_rc1.pdf, 2010.

[23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40–52.

[24] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[25] W. Xu, V. N. Venkatakrishnan, R. Sekar, and I. V. Ramakrishnan, "A framework for building privacy-conscious composite web services," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[26] S. Chong, K. Vikram, and A. C. Myers, "Sif: enforcing confidentiality and integrity in web applications," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007, pp. 1:1–1:16.

[27] A. Krishnamurthy, A. Mettler, and D. Wagner, "Fine-grained privilege separation for web applications," in *WWW*, 2010, pp. 551–560.

[28] A. Mettler, D. Wagner, and T. Close, "Joe-e: A security-oriented subset of java," in *17th Network and Distributed System Security Symposium*, 2010.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *WWW*, 2010, pp. 921–930.

[30] S. Maffeis, J. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *IEEE Symposium on Security and Privacy*, 2010.

[31] C. Queinnec, "Inverting back the inversion of control or, continuations versus page-centric programming," in *Newsletter of ACM SIGPLAN Notices*, 2003, pp. 57–64.

[32] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web programming without tiers," in *Proceedings of 5th International Symposium on Formal Methods for Components and Objects (FMCO)*, 2006, pp. 266–296.

APPENDIX

A. THE HOOKS OF ZEND ENGINE

The Zend Engine provides a number of hooks, allowing developers to insert their own logic into the compilation and execution process. We list some of its hooks in the following:

- `zend_compile_file`: compiler hook, used to add addi-

tional logic to the compilation process, or override the existing compilation process.

- `zend_execute`: execution hook, used to add additional logic to the execution process, or override the existing execution process.
- `statement_handler`: invoked after executing each single PHP statement, mainly used for debugging.
- `fcall_begin_handler`: invoked before the execution enters each single function.
- `fcall_end_handler`: invoked before the execution returns from a function.
- `startup`: invoked when the zend extension is loaded at server setup.
- `shutdown`: invoked when the zend extension is unloaded at server shutdown.
- `activate`: invoked at the beginning of each request.
- `deactivate`: invoked at the end of each request.

Let us take the `zend_compile_file` and `zend_execute` hooks as examples. Assuming that we want to include some extra logic before and after PHP compiles a function, we can do the following:

```
old_compile_file = zend_compile_file;
zend_compile_file = my_compile_file;

zend_op_array *my_compile_file()
{
    ... Our logic before compilation ...

    // run the original compilation process
    zend_op_array *op_array op_array = old_compile_file();

    ... Our logic after compilation ...
}
```

We can do similar things to the execution process, i.e., adding extra logic before and after a program is executed.

```
old_execute = zend_execute;
zend_execute = my_execute;

my_execute()
{
    ... Our logic before execution ...

    // run the original execution process
    old_execute();

    ... Our logic after execution ...
}
```

B. DATABASE RING CONFIGURATION

The `GRANT` command in MySQL assigns specific privileges to users. In our SCUTA implementation, we use this command to assign rings to database objects. To avoid mistakes caused by manually running the `GRANT` command, we let users write their ring configuration into a file using an intuitive format; we have created a configuration tool to automatically convert the configuration into a series of `GRANT` commands. The format of the configuration file is described in the following:

```
[USER]
    Ring:Operations:Table:Columns
```

We describe three examples of ring configuration and their corresponding `GRANT` commands.

Permission on tables. Assuming `TableA`, `TableB` and `TableC` can be accessed by dbuser. We would like to to place `TableA` in ring 0, `TableB` in ring 1, and `TableC` in ring 2. We have the following configuration file:

```
[dbuser]
   0:ALL:TableA:*
   1:ALL:TableB:*
   2:ALL:TableC:*
```

Because SCUTA supports hierarchical structure, if we place a table in ring k, the tables can also be accessed from rings 0 to k. Our tool converts the above configuration file into the following commands:

```
GRANT ALL ON TableA TO dbuser_0;
GRANT ALL ON TableB TO dbuser_0;
GRANT ALL ON TableC TO dbuser_0;
GRANT ALL ON TableB TO dbuser_1;
GRANT ALL ON TableC TO dbuser_1;
GRANT ALL ON TableC TO dbuser_2;
```

Permission on columns. Assuming `MyTable` can be accessed by dbuser. We would like to place its columns into different rings: columns `Deadline` and `Action` in ring 0, column `Profile` in ring 1, and column `Name` in ring 2. We have the following configuration file:

```
[dbuser]
   0:ALL:MyTable:Deadline, Action
   1:ALL:MyTable:Profile
   2:ALL:MyTable:Name
```

Our tool converts the above configuration file into the following commands:

```
GRANT ALL (Deadline, Action) ON MyTable TO dbuser_0;
GRANT ALL (Profile, Name) ON MyTable TO dbuser_0;
GRANT ALL (Profile, Name) ON MyTable TO dbuser_1;
GRANT ALL (Name) ON MyTable TO dbuser_2;
```

Permission on operations. Assuming `MyTable` can be accessed by `dbuser`, and it has a column called `Profile`. We would like to assign different operation privileges to this column: ring 2 can conduct `SELECT` only (i.e. read-only), ring 1 can additionally conduct `UPDATE`, and ring 0 can additionally conduct `DELETE` and `INSERT`. We have the following configuration file:

```
[dbuser]
   0:DELETE, INSERT:MyTable:Profile
   1:UPDATE:MyTable:Profile
   2:SELECT:MyTable:Profile
```

Our tool converts the above configuration file into the following commands:

```
GRANT DELETE (Profile) ON MyTable TO dbuser_0;
GRANT INSERT (Profile) ON MyTable TO dbuser_0;
GRANT UPDATE (Profile) ON MyTable TO dbuser_0;
GRANT SELECT (Profile) ON MyTable TO dbuser_0;
GRANT UPDATE (Profile) ON MyTable TO dbuser_1;
GRANT SELECT (Profile) ON MyTable TO dbuser_1;
GRANT SELECT (Profile) ON MyTable TO dbuser_2;
```