acmqueue Weapons of Mass Assignment

A Ruby on Rails app highlights some serious, yet easily avoided, security vulnerabilities.

Patrick McKenzie, Kalzumeus

In May 2010, during a news cycle dominated by users' widespread disgust with Facebook privacy policies, a team of four students from New York University published a request for \$10,000 in donations to build a privacy-aware Facebook alternative. The software, Diaspora, would allow users to host their own social networks and own their own data. The team promised to open-source all the code they wrote, guaranteeing the privacy and security of users' data by exposing the code to public scrutiny. With the help of front-page coverage from the *New York Times*, the team ended up raising more than \$200,000. They anticipated launching the service to end users in October 2010.

On September 15, Diaspora (<https://joindiaspora.com/>) released a "pre-alpha developer preview" of its source code (<https://github.com/diaspora/diaspora>). I took a look at it, mostly out of curiosity, and was struck by numerous severe security errors. I spent the next day digging through the code locally and trying to get in touch with the team to address them, privately. The security errors were serious enough to jeopardize the goals of the project.

This article describes the mistakes that compromised the security of the Diaspora developer preview. Avoiding such mistakes via better security practices and better choice of defaults will make applications more secure.

DIASPORA ARCHITECTURE

Diaspora is written against Ruby on Rails 3.0, a popular modern Web framework. Most Rails applications run as very long-lived processes within a specialized Web server such as Mongrel or Thin. Since Rails is not threadsafe, typically several processes will run in parallel on a machine, behind a threaded Web server such as Apache or nginx. These servers serve requests for static assets directly and proxy dynamic requests to the Rails instances.

Architecturally, Diaspora is designed as a federated Web application, with user accounts (seeds) collected into separately operated services (pods), in a manner similar to e-mail accounts on separate mail servers. The primary way end users access their Diaspora accounts is through a Web interface. Pods communicate with each other using encrypted XML messages.

Unlike most Rails applications, Diaspora does not use a traditional database for persistence. Instead, it uses the MongoMapper ORM (object-relational mapping) to interface with MongoDB, which its makers describe as a "document-oriented database" that "bridges the gap between key/value stores and traditional relational databases." MongoDB is an example of what are now popularly called NoSQL databases.

While Diaspora's architecture is somewhat exotic, the problems with the developer preview release stemmed from very prosaic sources.

SECURITY IN RUBY ON RAILS

Web application security is a very broad and deep topic, and is treated in detail in the official Rails security guide (<http://guides.rubyonrails.org/security.html>) and the OWASP (Open Web Application Security Project) list of Web application vulnerabilities (http://www.owasp.org/index.php/Top_10_2007), which would have helped catch all of the issues discussed in this article. While Web application security might seem overwhelming, the errors discussed here are elementary and can serve as an object lesson for those building public-facing software.

A cursory analysis of the source code of the Diaspora prerelease revealed on the order of a half-dozen critical errors, affecting nearly every class in the system. There were three main genres, detailed below. All code samples pulled from Diaspora's source at launch [note: I have forked the Diaspora public repository on GitHub and created a tag so that this code can be examined: https://github.com/patio11/diaspora/tree/diaspora_launch] were reported to the Diaspora team immediately upon discovery, and have been reported by the team as fixed.

AUTHENTICATION!=AUTHORIZATION: THE USER CANNOT BE TRUSTED

The basic pattern in the following code was repeated several times in Diaspora's code base: security-sensitive actions on the server used parameters from the HTTP request to identify pieces of data they were to operate on, without checking that the logged-in user was actually authorized to view or operate on that data.

```
#photos_controller.rb
def destroy
  @album = Album.find_by_id params[:id] # No authorization check.
  @album.destroy
  flash[:notice] = "Album #{@album.name} deleted."
  respond_with :location => albums_url
end
```

For example, if you were logged in to a Diaspora seed and knew the ID of any photo on the pod, changing the URL of any destroy action visible to include the ID of any other user's photo would let you delete that second photo. Rails makes such exploits very easy, since URLs to actions are trivially easy to guess, and object IDs "leak" all over the place. Do not assume that an object ID is private.

Diaspora, of course, does attempt to check credentials. It uses Devise, a library that handles authentication, to verify that you can get to the destroy action only if you are logged in. As shown in the code example above, however, Devise does not handle authorization—checking to see that you are, in fact, permitted to do the action you are trying to do.

IMPACT

When Diaspora shipped, an attacker with a free account on any Diaspora node had, essentially, full access to any feature of the software vis-à-vis someone else's account. That is quite a serious vulnerability, but it combines with other vulnerabilities in the system to allow attackers to commit more subtle and far-reaching attacks than merely deleting photos.

HOW TO AVOID THIS

Check authorization prior to sensitive actions. The easiest way to do this (aside from using a library to handle it for you) is to take your notion of a logged-in user and access user-specific data only through that. For example, Devise gives all actions access to a `current_user` object, which is a stand-in for the currently logged-in user. If an action needs to access a photo, it should call `current_user.photos.find(params[:id])`. If a malicious user has subverted the `params` hash (which, since it comes directly from an HTTP request, must be considered “in the hands of the enemy”), that code will find no photo (because of how associations scope to the `user_id`). This will instantly generate an ActiveRecord exception, stopping any potential nastiness before it starts.

MASS ASSIGNMENT WILL RUIN YOUR DAY

We have learned that if we forget authorization, then a malicious user can do arbitrary bad things to people. In the following example, since the user update method is insecure, an attacker could meddle with their profiles. But is that all we can do?

```
#users_controller.rb
def update
  @user = User.find_by_id params[:id] # <-- No authorization check.
  prep_image_url(params[:user])

  @user.update_profile params[:user] # <-- Pass untrusted input to @user then...
  respond_with(@user, :location => root_url)
end

#user.rb
def update_profile(params)
  if self.person.update_attributes(params) # <-- insert input directly to DB.
    #omitted for clarity
  end
end
```

Unseasoned developers might assume that an update method can only update things on the Web form prior to it. For example, the form shown in figure 1 is fairly benign, so one might think that all someone can do with this bug is deface the user’s profile name and e-mail address:

This is dangerously wrong.

Rails by default uses something called mass update, where `update_attributes` and similar methods accept a hash as input and sequentially call all accessors for symbols in the hash. Objects will update both database columns (or their MongoDB analogs) and will call `parameter_name=` for any `:parameter_name` in the hash that has that method defined.

IMPACT

Let’s take a look at the Person object in the following code to see what mischief this lets an attacker do. Note that instead of updating the profile, `update_profile` updates the Person: Diaspora’s internal

FIGURE 1

Profile Editing Form in Diaspora Preview

DIASPORA* PREVIEW

Family | Work | Search | edit profile | logout

All Aspects | manage

Editing profile

Picture

You don't have any photos! Go to the [albums](#) page to upload some.

Info

DIASPORA USERNAME: patio11@localhost

First name

Last name

Email

Cancel or Update Profile

notion of the data associated with one human being, as opposed to the login associated with one e-mail address (the User). Calling something `update_profile` when it is really `update_person` is a good way to hide the security implications of such code from a reviewer. Developers should be careful to name things correctly.

#Person.rb

```
class Person
  #omitted for clarity
  key :url, String
  key :diaspora_handle, String, :unique => true
  key :serialized_key, String #Public/private key pair for encryption.

  key :owner_id, ObjectID #Extraordinarily security sensitive because...

  one :profile, :class_name => 'Profile'
  many :albums, :class_name => 'Album', :foreign_key => :person_id
  belongs_to :owner, :class_name => 'User' #... changing it reassigns account ownership!
```

end

#User.rb

```
one :person, :class_name => 'Person', :foreign_key => :owner_id
```

This means that by changing a Person's `owner_id`, one can reassign the Person from one account (User) to another, allowing one not only to deny arbitrary victims their use of the service, but

also to take over their accounts. This allows the attacker to impersonate them, access their data at will, etc. This works because the “one” method in MongoDB picks the first matching entry in the DB it can find, meaning that if two Persons have the same `owner_id`, the owning User will nondeterministically control one of them. This lets the attacker assign your `Person#owner_id` to be his `#owner_id`, which gives the attacker a 50-50 shot at gaining control of your account.

It gets worse: since the attacker can also reassign his own data’s `owner_id` to a nonsense string, this delinks his personal data from his account, which will ensure that his account is linked with the victim’s personal data.

It gets worse still. Note the `serialized_key` column. If you look deeper into the User class, that is its serialized public/private encryption key pair. Diaspora seeds use encryption when talking with each other so that the prying eyes of Facebook can’t read users’ status updates. This is Diaspora’s core selling point. Unfortunately, an attacker can use the combination of unchecked authorization and mass update *silently to overwrite the user’s key pair*, replacing it with one the user generated. Since the attacker now knows the user’s private key, regardless of how well implemented Diaspora’s cryptography is, the attacker can read the user’s messages at will. This compromises Diaspora’s core value proposition to users: that their data will remain safe and in their control.

This is what kills most encryption systems in real life. You don’t have to beat encryption to beat the system; you just have to beat the weakest link in the chain around it. That almost certainly isn’t the encryption algorithm—it is probably some inadequacy in the larger system added by a developer in the mistaken belief that strong cryptography means strong security. Crypto is not soy sauce for security.

This attack is fairly elementary to execute. It can be done with a tool no more complicated than Firefox with Firebug installed: add an extra parameter to the form, switch the submit URL, and instantly gain control of any account you wish. Of particular note to open source software projects and other scenarios where the attacker can be assumed to have access to the source code, this vulnerability is very visible: the controller in charge of authorization and access to the user objects is a clear priority for attackers because of the expected gains from subverting it. A moderately skilled attacker could find this vulnerability and create a script to weaponize it in a matter of minutes.

HOW TO AVOID THIS

This particular variation of the attack could be avoided by checking authorization, but that does not by itself prevent all related attacks. An attacker can create an arbitrary number of accounts, changing the `owner_id` on each to collide with a victim’s legitimate user ID, and in doing so successfully delink the victim’s data from his or her login. This amounts to a denial of service attack, since the victim loses the utility of the Diaspora service.

After authentication has been fixed, write access to sensitive data should be limited to the maximum extent practical. A suitable first step would be to disable mass assignment, which should always be turned off in a public-facing Rails app. The Rails team presumably keeps mass assignment on by default because it saves many lines of code and makes the 15-minute blog demo nicer, but it is a security hole in virtually all applications.

Luckily, this is trivial to address: Rails has a mechanism called `attr_accessible`, which makes only the listed model attributes available for mass assignment. Allowing only safe attributes to be mass-assigned (for example, data you would expect the end users to be allowed to update, such as their names rather than their keys) prevents this class of attack. In addition, `attr_accessible` documents

programmers' assumptions about security explicitly in their application code: as a whitelist, it is a known point of weakness in the model class, and it will be examined thoroughly by any security review process.

This is extraordinarily desirable, so it's a good idea for developers to make using `attr_accessible` compulsory. This is easy to do: simply call `ActiveRecord::Base.attr_accessible(nil)` in an initializer, and all Rails models will automatically have mass assignment disabled until they have it explicitly enabled by `attr_accessible`. Note that this may break the functionality of common Rails gems and plugins, because they sometimes rely on the default. This is one way in which security is a problem of the community.

An additional mitigation method, if your data store allows it, is to explicitly disallow writing to as much data as is feasible. There is almost certainly no legitimate reason for `owner_id` to be reassignable. ActiveRecord lets you do this with `attr_readonly`. MongoMapper does not currently support this feature, which is one danger of using bleeding-edge technologies for production systems.

NOSQL DOESN'T MEAN NO SQL INJECTION

The new NoSQL databases have a few decades less experience getting exploited than the old relational databases we know and love, which means that countermeasures against well-understood attacks are still immature. For example, the canonical attack against SQL databases is SQL injection: using the user-exposed interface of an application to craft arbitrary SQL code and execute it against the database.

```
def self.search(query)
  Person.all('$where' => "function() { return this.diaspora_handle.match(/^#{query}/i) ||
    this.profile.first_name.match(/^#{query}/i) ||
    this.profile.last_name.match(/^#{query}/i); }") #Permits code injection to MongoDB.
end
```

IMPACT

The previous code snippet allows code injection into MongoDB, effectively allowing an attacker full read access to the database, including to serialized encryption keys. Observe that because of the magic of string interpolation, the attacker can cause the string including the JavaScript to evaluate to virtually anything the attacker desires. For example, the attacker could inject a carefully constructed JavaScript string to cause the first regular expression to terminate without any results, then execute arbitrary code, then comment out the rest of the JavaScript.

We can get one bit of data about any particular person out of this find call—whether the person is in the result set or not. Since we can construct the result set at will, however, we can make that a very significant bit. JavaScript can take a string and convert it to a number. The code for this is left as an exercise for the reader. With that JavaScript, the attacker can run repeated find queries against the database to do a binary search for the serialized encryption key pair:

“Return Patrick if his serialized key is more than 2^{512} . OK, he isn't in the result set? Alright, return Patrick if his key is more than 2^{256} . He is in the result set? Return him if his key is more than $2^{256} + 2^{255}$ . ...”

A key length of 1,024 bits might strike a developer as likely to be very secure. If we are allowed to do a binary search for the key, however, it will take only on the order of 1,000 requests to discover the key. A script executing searches through an HTTP client could trivially run through 1,000 accesses in a minute or two. Compromising the user's key pair in this manner compromises all messages the user has ever sent or will ever send on Diaspora, and it would leave no trace of intrusion aside from an easily overlooked momentary spike in activity on the server. A more patient attacker could avoid leaving even that.

This is probably not the only vulnerability caused by code injection. It is very possible that an attacker could execute state-changing JavaScript through this interface, or join the Person document with other documents to read out anything desired from the database, such as user password hashes. Evaluating whether these attacks are feasible requires in-depth knowledge of the internal workings of MongoDB and the Ruby wrappers for it. Typical application developers are insufficiently skilled to evaluate parts of the stack operating at those levels: it is essentially the same as asking them whether their SQL queries would allow buffer overruns if executed against a database compiled against an exotic architecture. Rather than attempting to answer this question, sensible developers should treat any injection attack as allowing a total system compromise.

HOW TO AVOID THIS

Do not interpolate strings in queries sent to your database. Use the MongoDB equivalent of prepared statements. If your database solution does not have prepared statements, then it is insufficiently mature to be used in public-facing products.

BE CAREFUL WHEN RELEASING SOFTWARE TO END USERS

One could reasonably ask whether security flaws in a developer preview are an emergency or merely a footnote in the development history of a product. Owing to the circumstances of its creation, Diaspora never had the luxury of being both publicly available but not yet exploitable. As a highly anticipated project, Diaspora was guaranteed to (and did) have publicly accessible servers available within literally hours of the code being available.

People who set up servers should know enough to evaluate the security consequences of running them. This was not the case with the Diaspora preview: there were publicly accessible Diaspora servers where any user could trivially compromise the account of another user. Moreover, even if one assumes that the server operators understand what they are doing, their users and their users' friends who are invited to join "The New Secure Facebook" are not capable of evaluating their security on Diaspora. They trust that, since it is on their browser and endorsed by a friend, it must be safe and secure. (This is essentially the same process through which they joined Facebook prior to evaluating the privacy consequences of that action.)

The most secure computer system is one that is in a locked room, surrounded by armed guards, and powered off. Unfortunately, that is not a feasible recommendation in the real world: software needs to be developed and used if it is to improve the lives of its users. Could Diaspora have simultaneously achieved a public-preview release without exposing end users to its security flaws? Yes. A sensible compromise would have been to release the code with the registration pages elided, forcing developers to add new users only via Rake tasks or the Rails console. That would preserve 100

percent of the ability of developers to work on the project and for news outlets to take screenshots—without allowing technically unsophisticated people to sign up on Diaspora servers.

The Diaspora community has taken some steps to reduce the harm of prematurely deploying the software, but they are insufficient. The team curates a list of public Diaspora seeds (<https://github.com/diaspora/diaspora/wiki/>), including a bold disclaimer that the software is insecure, but that sort of passive posture does not address the realities of how social software spreads: friends recommend it to friends, and warnings will be unseen or ignored in the face of social pressure to join new sites.

COULD RAILS HAVE PREVENTED THESE ISSUES?

Many partisans for languages or framework argue that “their” framework is more secure than alternatives and that some other frameworks are by nature insecure. Insecure code can be written in any language: indeed, given that the question “Is this secure or not?” is algorithmically undecidable (it trivially reduces to the halting problem), one could probably go so far as to say it is flatly impossible to create any useful computer language that will always be secure.

That said, defaults and community matter. Rails embodies a spirit of convention over configuration, an example of what the team at 37signals (the original authors of Rails) describes as “opinionated software.” Rails conventions are pervasively optimized for programmer productivity and happiness. This sometimes trades off with security, as in the example of mass assignment being on by default.

Compromises exist on some of these opinions that would make Rails more secure without significantly impeding the development experience. For example, Rails could default to mass assignment being available in development environments, but disabled in production environments (which are, typically, the ones that are accessible by malicious users). There is precedent for this: for example, Rails prints stack traces (which may include sensitive information) only for local requests when in production mode, and gives less informative (and more secure) messages if errors are caused by nonlocal requests.

No amount of improving frameworks, however, will save programmers from mistakes such as forgetting to check authorization prior to destructive actions. This is where the community comes in: the open source community, practicing developers, and educators need to emphasize security as a process. There is no technological silver bullet that makes an application secure: it is made more secure as a result of detailed analysis leading to actions taken to resolve vulnerabilities.

This is often neglected in computer science education, as security is seen as either an afterthought or an implementation detail to be addressed at a later date. Universities often grade like industry: a program that operates successfully on almost all of the inputs scores almost all of the possible points. This mindset, applied to security, has catastrophic results: the attacker has virtually infinite time to interact with the application, sometimes with its source code available, and see how it acts upon particular inputs. In a space of uncountable infinities of program states and possible inputs, the attacker may need to identify only one input for which the program fails to compromise the security of the system.

It would not matter if everything else in Diaspora were perfectly implemented; if the search functionality still allowed code injection, that alone would result in total failure of the project’s core goals.

IS DIASPORA SECURE AFTER THE PATCHES?

Security is a result of a process designed to produce it. While the Diaspora project has continued iterating on the software, and is being made available to select end users as of the publication of this article, it is impossible to say that the architecture and code are definitely secure. This is hardly unique to Diaspora: almost all public-facing software has vulnerabilities, despite huge amounts of resources dedicated to securing popular commercial and open source products.

This is not a reason to despair, though: every error fixed or avoided through improved code, improved practices, and security reviews offers incremental safety to the users of software and increases its utility. We can do better, and we should start doing so. **Q**

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PATRICK MCKENZIE is the founder of Kalzumeus, a small software business in Ogaki, Japan. His main products—Bingo Card Creator (<http://www.bingocardcreator.com>) and Appointment Reminder (<http://www.appointmentreminder.org>)—are both written in Ruby. He graduated from Washington University with a BS/CS in computer science and a BA in East Asian studies in 2004.

© 2011 ACM 1542-7730/11/0300 \$10.00