CS259 Final Project: OpenID

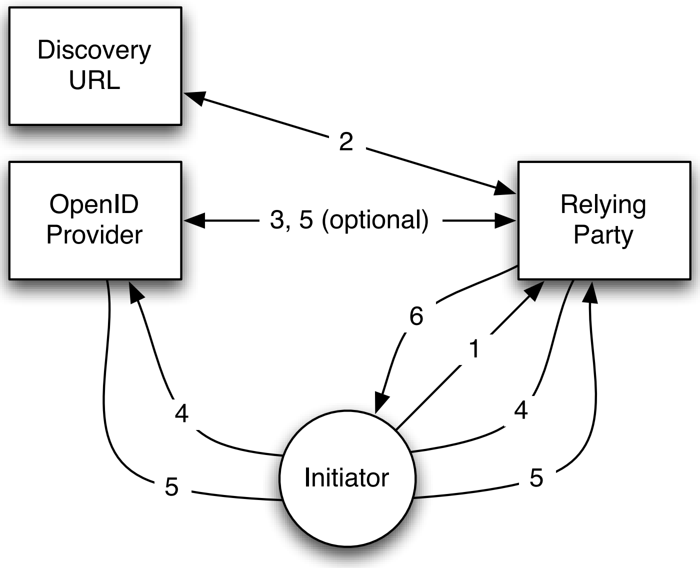
Ben Newman
Shivaram Lingamneni

Introduction

The OpenID protocol promises to consolidate web users' countless and easily forgotten passwords under a decentralized, single-signon authentication system. Many of the details of the protocol are left under-specified: in particular, the details of the login process, regarding the use or eschewal of password-based authentication and the mechanism for ensuring intent to authenticate, are considered "beyond the scope of this document" by the authors of the specification. Finding attacks on the protocol is not difficult (try to identify a simple phishing attack after reading the Protocol Overview section), but finding attacks that are within the scope of the protocol, as defined by its authors, presents a challenge. Our model of the protocol can be used to exhibit at least one intrinsic vulnerability, an instance of "session swapping."[1] Nevertheless our model is simple enough to allow exhaustive exploration, so we believe it occupies a non-trivial middle ground between completeness and ease of use.

Protocol overview

We have modeled the OpenID protocol up to the level of detail shown in the following diagram:



1. The Initiator (end user) sends an initiation message (1) to the Relying Party (interchangeably, the responder; i.e., some authenticating website) containing the user's identifier (a unique OpenID URL).
2. The Relying Party visits the URL provided by the user[2] to determine how to reach an OpenID Provider capable of verifying the user's identity.
3. The Relying Party and the Provider optionally establish a shared secret to permit secure direct communication.
4. The Relying Party issues an authentication request to the Provider by redirecting the Initiator's browser to the Provider's URL. The request contains the following fields:
   - mode ("immediate" or "setup")
   - claimed_id (the user-supplied identifier)
   - identity (the op-local identifier)

[1]A term coined by Adam Barth to identify a class of cross-site request forgery-based attacks that cause their victims to be inadvertently authenticated as a malicious party.

[2]Generally a simple XML document containing, among other data, the address of the Provider. Although OpenID supports multiple methods of Provider discovery, this flexibility was not interesting to model.

- assoc_handle (key identifying the shared secret established in step 3, if any)
- return_to (URL to which user should return after logging in)
- realm (typically the domain of the authenticating website)

The user now authenticates to the Provider via some secure mechanism. In practice, OpenID providers use password-based authentication and session cookies. If the user is not logged in to the Provider, he or she will see a password prompt. If login has already taken place, the user will not be prompted further. (The "mode" field tells the Provider if the Relying Party requires setup information, such as date of birth or gender. We did not model such exchanges of information.)

5. Depending on whether the authentication was successful, the Provider asserts approval or failure to the Relying Party. The assertion is sent indirectly through the Initiator's browser, in the query string of an HTTP redirect URL, and contains the following fields (in the positive case):

   - mode (value "id_res")
   - op_endpoint
   - claimed_id (as before)
   - identity (as before)
   - return_to (copied)
   - response_nonce (server time plus some salt)
   - signed (which fields were signed)
   - sig (signed fields)

   The Relying Party verifies the information contained in the authentication assertion by checking its signature, return URL, and nonce, using the shared key established in step 3 if possible or else sending a direct request to the Provider.

6. If the information is verified, the Relying Party considers the Initiator authenticated to the supplied OpenID. Typically, the RP will issue a session cookie to the Initiator. (The content and form of the cookie are not specified by the protocol.)

There are two things to note about the messages passed. One is that all messages are sent via HTTP or HTTPS; the protocol makes use only of existing mechanisms, in particular POST and HTTP redirect. Another is that if the RP and OP successfully share a secret in step 3, then there is no further direct communication between them. The authentication request and response messages are passed through the Initiator's browser on their way to their destination. The motivation for this is to make use of the authentication that exists between the client and the provider. In particular, if the Provider implements authentication using session cookies, the cookie will be sent along with the auth-request message. Analogously, since the Initiator is the final sender of the auth-response message, the RP can immediately issue a session cookie to him after the message is verified. As we will see, these indirect messages also have the effect of giving an adversary without network capabilities nontrivial man-in-the-middle power over the protocol.

Modeling

Simplifications

In modeling this protocol, we had to make several simplifying assumptions, in order to reduce the state space and make the protocol easier to model. We also had to decide on the capabilities we would allow our adversary to have.

One problem we dealt with was whether to model the connections as happening over SSL or not. The specification allows for the use of HTTP or HTTPS URIs as endpoints by OPs and RPs. However, since the use of digital signatures and nonces in the algorithm implies that the protocol is expected to be secure without any further encryption, we decided to assume that the entire protocol happens over HTTP.

We made several decisions that involved reducing the scope of our model. For one, we decided not to model the association step of the protocol, where the RP and OP share a secret. We reasoned that since association uses protocols generally considered secure (the specification suggests Diffie-Hellman with either SHA-1 or SHA-256), the introduction of those messages was unlikely to create a security vulnerability. (We don't have any formal justification for this assumption.)

We also did not model the mechanism by which users authenticate to their OPs (although in practice this is almost universally done by means of a session cookie), or any interactions subsequent to the processing of the authentication response message by the RP (again, typically the issuing of a session cookie). We had several motivations. One was that since the protocol leaves these unspecified, leaving them out would make our model more general. Another was that attacks on the security of session cookies seemed either trivial or impossible. On the one hand, a network attacker with eavesdropping capabilities can simply steal the session cookies. On the other, since cookies are subject to viewing and alteration only by the issuing site, we guessed that it would be impossible for an attacker without network capabilities to steal an OP or RP session cookie. Accordingly, we directed our attention elsewhere.

We furthermore eliminated modeling of the fields response_nonce and assoc_handle. It was an easy decision to eliminate assoc_handle, because the specification is almost silent on the topic, and we thought any modeling we did would have been implementation-specific. Without assoc_handle, our model effectively creates a new provider every time an association is made (i.e., there is no ability to model the relationship between two separate connections to the same provider), which we thought acceptable. Deciding to skip modelling of response_nonce was more difficult, and was connected to a broader problem: since OpenID providers and responders stay up for arbitrary lengths of time, how many transactions should our model cover? We ultimately decided to limit our model to a single login.

Finally, we decided on the following intruder model. An intruder controls a dummy provider and a dummy responder, the actions of which he controls. These dummies are capable of making associations with legitimate RPs and OPs. The intruder has no network capabilities, but can intercept, save, and replay messages addressed to him. Furthermore, the intruder can synthesize any message, as long as it respects the rules of digital signatures (the intruder can only sign a message with a shared secret if that secret-sharing has actually taken place, and if the intruder is privy to the secret). As mentioned before, network attackers can make trivial attacks on the system (an eavesdropper can steal the session cookie, and an active attacker can mimic the provider, or, better yet, substitute his own URL for the provider's), so we felt justified in excluding these capabilities from our model.

Technical details of the model are explained in comments in the Murphi code. It is worth noting here that it is a subtle point whether a variable that identifies an "agent" actually represents a URI, an IP address, or a claimed OpenID identifier.

It should be mentioned that these simplifying assumptions were finalized only after seeing Adam Barth's description of the XSRF session-swapping attack on OpenID. Before seeing it, we had made different assumptions; in a sense, part of our modeling process was the replication of this attack.

Compensating for Murphi's limitations

Our concern for keeping the state space tractable was somewhat confounded by Murphi's type system.

Murphi supports `union` types, so we initially designed a hierarchy of agent types that expressed not only the scalar types `ResponderId`, `ProviderId`, `InitiatorId`, and `IntruderId` but also pairwise `union`s of the first three with `IntruderId`, capturing the notion of potentially dishonest agents, and the `union` of all four types, which formed a supertype called `AgentId`.

There were three problems with this design, all stemming from the unfortunate fact that `union` types do not induce a subtyping relation:

- `union` types cannot be composed, as the components of a `union` type are restricted to scalar and enum types. Our hierarchy was therefore limited to a depth of one, so that our final model has only the basic scalar types and the `union` type `AgentId`.

- The `ismember` function cannot be used to test membership in a `union` type, so we were forced to use explicit disjunctions of `ismember` predicates over concrete types. It is worth acknowledging that the previous problem had already eliminated the intermediate types from our hierarchy, and of course it was never very useful useful to test membership in the universal `AgentId` type.

- Even when type `A` is a subset of type `B`, assignments from values of type `A` to fields or variables of type `B` are not allowed, so we had to modify the fields of our `Initiator`, `Responder`, and `Provider`, and `Intruder` record types to have type `AgentId` if ever the field could be assigned from more than one scalar type. Also, our global arrays of agents `ini`, `res`, and `pro` had to be indexed with the `AgentId` type, which increased their size by adding dummy indices.

We compensated for these limitations by pushing our type checking to run-time. In our code, you will frequently see variables quantified over `AgentId` accompanied by `ismember` disjunctions guarding the execution of a rule or the evaluation of an invariant. These guards were essential for pruning the state space and provided a moderately satisfactory replacement for the type hierarchy we had intended.

Results

Session swapping attack

Running the model checker on the model we define in "ns.m" (type "make" and then "./ns -ndl -tv") produces an violation of the "initiator correctly authenticated" invariant after searching approximately 2400 states (i.e., almost immediately). Here is that invariant:

```
invariant "initiator correctly authenticated"
  forall i: ResponderId do
    res[i].state = R_COMMIT &
    (!SUPPRESS_SWAPPING |
     ismember(res[i].initiator, InitiatorId))
    ->
    res[i].initiator = res[i].auth_party
  end;
```

The relevant portion of the final state is as follows. Note that the `initiator` field is veridical metadata that we have introduced, not part of the protocol.

```
res[ResponderId_1].state:R_COMMIT
res[ResponderId_1].initiator:IntruderId_1
res[ResponderId_1].provider:ProviderId_1
res[ResponderId_1].auth_party:InitiatorId_1
```

The intruder initiated the protocol by posting its OpenID to the responder, and the responder replied with an authentication request to be forwarded to the provider. The intruder "intercepts" this message, even though it was actually sent to the intruder, and "reroutes" it to the provider, just as an honest initiator would do. The provider replies with a positive authentication response, which the intruder intercepts.

Up to this point, the intruder has behaved exactly as an honest initiator would behave; however, if the intruder now injects a forged cross-site request into a victim's browser, it can effectively change the source of the provider's response, so that the reponse appears to have been forwarded through the unsuspecting user's browser. When the responder issues its authentication token, it does so for whatever `auth_party` submits this request; in this case, therefore, the unsuspecting user receives a cookie associated with the intruder's credentials and is thus inadvertently authenticated as the intruder.

We model the ability of the intruder to forge such a request by simply allowing an intruder to change the source of the message, provided the new source is either the intruder itself or some end user (it does not make sense that an intruder could forge requests from responders or providers).

Termination of search

If the constant `SUPPRESS_SWAPPING` is set to true, the previous attack will be ignored because the initiator of the authentication process is not in the set of honest initiators. We introduced this constant so that we could search for similar attacks, but we did not discover any.

This failure to find similar attacks was not simply a matter of running out of resources for exploring the state space. In fact, the exploration eventually terminates without violating any invariants. This result is stronger than an inconclusive "too many active states" outcome, and suggests that further restrictions of our model would not produce new attacks. Our concern for state space tractability paid off: the model checker explores 854076 states, firing 1904227 rules, before reporting that no error was found.[3]

Forced login attack

In conventional password-based login schemes, it is impossible to force someone to authenticate to a server without knowing their password. This provides a guard against XSRF attacks, since the user cannot be tricked into taking actions on websites to which he or she is not authenticated. However, OpenID lacks this guard. Consider the following situation. An XSRF-capable attacker injects the login form for some relying party, complete with the intended victim's OpenID identifier, into a page that the victim will view. (Notice that the only information needed to construct this form is the RP's endpoint and the victim's USI, both of which are public.) Upon submission of this form, the victim, who is already authenticated to his or her OP, becomes authenticated to the RP. Now, if there are further XSRF vulnerabilities on the RP's site, the attacker can use further code insertions to take more actions using the victim's identity.

Note that this attack cannot be prevented by the RP's use of protections against cross-site submission of the login form. To get around these, the attacker can submit the login form (using the victim's USI) on the RP's website (or via a user agent of his own that forges the referrer field). Then, the attacker intercepts the auth-request message that comes back and injects it, with a redirect to the provider, into a page the victim will see. The victim ends up submitting the auth-request form to the OP, being authenticated, and being redirected back to the RP.

Since silent authentication (i.e., authentication not requiring action by the end user) is a feature of OpenID, it is difficult to prevent this attack using the existing framework. The good news is that Adam Barth's patch for the session swapping attack (an additional cookie, set during submission of the login form, that ensures that the sender of the login form and the sender of the auth-response message are the same), together with protections against cross-site submission of the login form, solves this issue.

Other directions

Intent to authenticate

We considered adding a boolean field to the initiator record to model initiators who do not wish to log in. We could have changed the provider rules so that a negative rather than positive authentication request was sent back to the relying party when the apparent initiator did not intend to be authenticated at all. In the end, we concluded this preventative measure was not realistic enough to justify the complication, but there is clearly further opportunity to investigate the dynamics of intent (perhaps by introducing per-provider intent).

Increasing the scope

As mentioned before, the present model simulates a single run of the protocol. It also ignores certain aspects of the protocol, notably the management of multiple associations between RPs and OPs and the use of a nonce against replay attacks. All these areas could be the subject of further exploration using the methods we have described here.

It might be worthwhile to examine some prominent implementations of the specification (e.g., LiveJournal) and see if details left out of the specification can be filled in and modelled.

Realm spoofing

It is tempting to imagine that the relying party has nothing to gain from dishonesty: either it denies authentication at the risk of alienating a valid user, or it allows authentication at the risk of giving a malicious party unintended control over privileged information.

[3] This takes about three minutes on a myth, so we do not advise using the elaines.

One way of examining this assumption is to ask what information is witheld from the relying party during an honest execution of the protocol. Perhaps the most sensitive item exchanged beyond the grasp of the relying party is the end user's login information; in some sense the whole purpose of the protocol is to permit users to enter their credentials on a single, trusted site. A dishonest relying party can steal this information via phishing (by simply redirecting the user to a phony provider), but this attack, while noteworthy, is intrinsic to password-based authentication, and the authors of the protocol are right to argue that passwords are not the only method of verifying a claimed identity.

Another set of sensitive data is stored by the provider and may be revealed to the relying party at the discretion of the end user. The second version of the OpenID protocol allows arbitrary personal information—full name, date of birth, location, *et cetera*—to be communicated back to the relying party after login, as the user desires. While the relying party cannot effectively spoof the return-to address, it may provide a "realm" instead, which has no function in the protocol except to provide information about the site with which the user is deciding whether or not to authenticate. If this information is misrepresented, the user may be induced to provide more personal information than he or she intends. It remains to be seen how sensitive this information will be in practice, but the protocol assumes nearly absolute trust of the provider, so realm spoofing poses a considerable threat to the user's trust.

Conclusion

In 1993 cartoonist Peter Steiner observed, "On the internet, nobody knows you're a dog." Arguably more important than hiding such information is the ability to prove otherwise—that one is neither a dog, nor a malicious party, nor under the control of a malicious party. In this regard the OpenID protocol leaves much to be desired, but it does provide a platform around which best practices can accumulate, such as using a nonce to track the initiator of an authentication request all the way to the end of the process, or avoiding inadvertent authentication by deprecating the all-too-convenient "immediate" mode. If over-specification and vulnerability go hand in hand, the modesty of the OpenID protocol may be prudent, but better tools for anticipating XSRF vulnerabilities should make it easier to move confidently toward a more fully specified protocol.

Acknowledgements

We would like to thank Adam Barth for his help with XSRF and session swapping attacks, as well as Arnab Roy for his help with Murphi.