8. Security

Many of Joule's security foundations were drawn from or inspired by KeyKOS, a capability-based operating system that provides NCSC (National Computer Security Center) B3-level security. Security can be thought of as the extreme of modularity: truly independent programs can only interact with each other through explicit and controlled boundaries.

Security, like modularity, is first supported by negative capabilities: operations that are prevented. After insecure abilities have been removed (such as the ability to write any file or write to any memory location), and the system has been reduced to secure foundations, one then builds abstractions that provide all the standard functionality of insecure systems without exposing programs to risk. Finally, one establishes tools and methodologies with which programs can securely engage in otherwise risky activities. This methodology for security is really a methodology for managing and arranging trust relationships.

This section first describes encapsulation, the enforcement of rules for accessibility and visibility. Encapsulation makes each server inviolable by other servers—the only thing a client can do to a server is send it messages to which the server explicitly decides how to respond.

Encapsulation brings with it polymorphism and anonymity: servers can only be distinguished by how they behave, so servers could be written that pretend to be other servers (for instance, money). How does one build trust relationships in such a system? We describe the technique used in Joule with which servers can prove their identity to each other, allowing the establishment and extension of trust relationships between servers. This supports the creation of extended networks of servers that cooperate and subcontract with each other.

With the establishment of these networks, the encapsulation and trust issues arise all over again: does the original client trust a subcontractor? There are many properties of a server that are composed from their subcontractors. The two we describe here are discretion ("Will the server keep secrets?") and durability ("Will the server still be in operation in the future?"). Other such properties, like timeliness and robustness, are not explored here.

8.1. Encapsulation

Encapsulation is what people commonly think of when they think about security. It is the enforcement of rules for accessibility and visibility. Languages like C and C++ provide weak modularity because any program in the same address space can convert a number to a pointer and violate the integrity of other parts of the code. Encapsulation allows programs to run without interference or corruption from other programs.

8.1.1. Capability Security

Capability security is the foundation for good encapsulation. This section describes capability security. Certain powerful capabilities, particularly those that violate encapsulation, such as the ability to read and write any section of memory, are closely held by severely restricted service providers.

8.1.2. Accessibility Relationships

The semantics of the accessibility relationships from the abstract execution model constrain the kinds of communication and access that can happen among programs in a Joule system. The rules guarantee that all communication is by message passing, and that all access to a receiver is only through message passing. This guarantees the encapsulation of Joule programs.

8.2. Certification

To build trust among unknown servers requires the ability to prove their identities to each other. The identity might be of a type: servers that charge money want to be paid with *real* money (also implemented as a server), not a forged server that responds to the same messages as money. The identity might also be of a particular server: when sending to requests to a bank, a server wants it to be the bank at which it has its account, not just any bank (even though all banks run the same code).

8.2.1. Verifiers

Joule provides the `Verifier` abstraction, a mechanism for certification built using only encapsulation and message passing. Unlike KeyKOS brands, `Verifiers` require no support in the computational model.

For more about `SealedEnvelopes`, see Appendix D, *Energetic Secrets*.

`Verifiers` provide the services of a single-key encryption scheme using encapsulation. Given a value to be sealed, a `Verifier` will create and reveal a `SealedEnvelope` containing the supplied value that can only be opened by the `Verifier` that sealed the envelope. That `SealedEnvelope` can then be passed through insecure channels to some other server which has access to the same `Verifier`. That other server can open the `SealedEnvelope` and use the contained server.

Type `Verifier`
super `Basic`
seal the supplied contents in an envelope that can only be opened by the receiving Verifier, and reveal that sealed envelope.
op `seal: contents enveloped>`

```
given an envelope sealed by the receiving Verifier, reveal the contents of the envelope.
op unseal: envelope content>

The type for envelopes used by Verifiers. SealedEnvelopes have no other behavior
        (beyond Basics) than the secure access that Verifiers use to get at the
        contents. Verifiers are built using more primitive Verifiers, so no user program
        can get at the contents of a SealedEnvelope; only the proper Verifier can.
Type SealedEnvelope
super Basic

Only Verifiers can get at the private channel of a SealedEnvelope (because they prove
        their identity using other Verifiers).
op private: private>
```

`Verifiers` rely on encapsulation for their certification properties: unsealing a message proves that the originating party had access to the same `Verifier` as the receiver; with encapsulation, the receiver can know the extent in which the `Verifier` is visible, and so can know what code generated the message.

Here is the replacement code for the hierarchical bank account example (Chapter 6) that uses `Verifiers` to allow accounts access to each other's `Private` channel without exposing the `Private` channels to outside code. This line of code is added within the definition of the `make-account` server; it creates a new `Verifier` named `AccountPrivate` within the scope of the `Account` implementation:

```
Define AccountPrivate (make-verifier ::)
```

The remainder of the code replaces the corresponding insecure implementations in the original Account implementation code:

```
Reveal an Envelope containing the private channel for the receiver.
    op private: priv>
        AccountPrivate seal: Private priv>
Reveal the private channel of another account by asking it for a sealed Private channel
        and unsealing it.
    Server private :: account priv>
        Define box account private:
        AccountPrivate unseal: box priv>
```

The redefinition of the `private:` method now reveals a `SealedEnvelope` containing the `Private` channel rather than revealing the actual `Private` channel. The redefined `private` server (the internal server that an account calls to access the `Private` channel of another account) also uses that `Verifier`: it asks the desired account for an envelope containing the account's Private channel, then unseals that envelope using the `AccountPrivate` Verifier (the one shared by all accounts made by this `make-account` server.)

Both the account requesting a private channel and the account providing the private channel can be assured that the other is a real account, and can be assured that the `Private` channel is secure (not exposed to eavesdroppers or forgers). The type-authenticity (proving the other is a real account) is guaranteed because both parties know the other party has access to the `AccountPrivate Verifier` (or the seal/unseal wouldn't have worked), both parties know that the accessibility rules of the language guarantee that unless the make-account code reveals the

AccountPrivate Verifier, then only the body of that code can use it, and finally both parties know that their implementation in make-account doesn't reveal the AccountPrivate Verifier. The combination of these means that only an account could have provided a SealedEnvelope openable by AccountPrivate, and only an account could open that envelope. This means that messages sent on any account Private channel are sent by an account server, and so are part of the proper implementation of the Account abstraction.

An eavesdropper is a forwarder that wraps the SealedEnvelope in order to get access to the contained Private channel when the envelope is unsealed, or to engage in the protocol with which envelope is unsealed. At no time is the Private channel of an account exposed outside of the context of the account implementation, except in a SealedEnvelope that can only be opened in the Account implementation context. The only potential for exposure of the contents of the SealedEnvelope occurs during the unsealing operation; during the sealing, the account has an internal (i.e., no eavesdroppers) channel to the Private channel.

The unsealing operation uses the same kind of protocol in order to unseal the envelope. All Verifiers and SealedEnvelopes within a trust boundary share access to a single Verifier (the implementation of which distributes efficiently) through which they can securely connect with no eavesdroppers. As a result, during the unseal operation, the AccountPrivate Verifier and the SealedEnvelope containing the desired private channel make a secure connection through which the Envelope reveals the contained Private channel to the Verifier which then reveals it to the caller of unseal:.

8.2.1.1. Unique Tokens

In the private protocol, an implementation could define messages to reveal information unique to each instance in order to prove identity. This code implements a simple unique token scheme in which the clients can only compare unique tokens, but cannot otherwise find out *anything* about them:

```
Server make-token
var myNext 0
var TokenPrivate (make-verifier :)
this should be a const declaration, but we don't have those yet.
Define the message used for procedure call (to make a unique token). This could also
        have been part of the first clause, but this is a better style for procedures with
        changing variables.
op : token>
    token> -> token
Here is the behavior for an instance made by make-token.
    Server token
    var myID myNext
Get the number of the other guy and then compare it against the number of the
        receiver.
    op = other equal?>
        Define hisNum (TokenPrivate unseal: (other private:))
        myNum = hisNum equal?>
 Reveal an Envelope containing the number unique to each instance.
    op private: envelope>
        TokenPrivate seal: myNum envelope>
set myNext myNext + 1
```

Tokens each contain a number guaranteed to be unique with the simple expedience of allocating them sequentially. The sequential ordering won't reveal anything about the running of the program even if the token creator is shared among untrusting programs because the numbers are never revealed except inside the actual implementation (just like `account` above).

8.3. Discretion

As described in the introduction to security, once a system has certification and encapsulation, it is possible to build large networks of servers that subcontract with each other to provide services to their clients. The naïve expansion to these large networks of servers leaves systems with the same precarious lack of robustness characteristic of networks today. Joule supports these large networks by allowing servers to establish and verify transitive properties of these subcontracting relationships so that a server can ensure its robustness. Many of these properties are managed through the same general mechanisms, but they will first be explored in the context of discretion. The techniques we describe here are called assurance by construction, assurance by auditing, and assurance by special execution.

Discretion control is a generalization of the better-known confinement problem. Successful *confinement* allows untrusted code to be run on private data without the untrusted code being able to communicate any secrets to the outside world. As the name implies, solutions to the confinement problem typically rely on running the untrusted code in a box out of which it cannot communicate. Successful *discretion control* allows that same untrusted code to communicate with other services, but still prevents secrets from being leaked.

8.3.1. Assurance by Construction

Assurance by construction means having mutually trusted third-parties build services for each other. The clients of such a construction service can then know that no other server has access to their private services, and so even if those services cheat, they still can't communicate secrets to the outside world. This section will describe the process in more detail, including how Factories can provide assurance by construction.

8.3.2. Assurance by Auditing

Auditors might be equipped with an abstraction-breaking tool which can examine an existing server for capabilities to steal secrets. This tool is very closely held. This is assurance by *auditing.* An auditor is able to audit a service for connections that could lead to data leaks or for dependencies on facilities that may be insufficiently permanent.

Modules pass an audit if they are functionally discreet. Factories produce modules that are structurally discreet—structural discretion can only be supplied by construction because a module that would otherwise be discreet might be shared with an indiscreet observer (a discreet database, for instance).

8.3.3. Assurance by Special Execution

Another technique involves an elaboration of the execution model with which the owner of a secret can send messages involving the secret inside special query messages. The ultimate effects of a query are solely upon those things named in the message. This may sound like a very limiting style of programming but such a query may, for instance, create a database and reveal access to it while guaranteeing that no one else has access. That the database was created with a query ensures that secrets entrusted to it are safe from disclosure. This is assurance by *special execution*.

8.4. Durability

Durability is the property of a service that it can guarantee its own survival and continued function. It is *durable* if no untrusted authority has the ability to destroy it or other services on which it depends. This can extend into very physical realms of assuring that communication links are independently redundant so that a single failure doesn't partition the network.

8.4.1. Implementation

Support for durability relies on the same foundations as discretion, but uses the mechanisms somewhat differently. This section will describe the differences.

8.4.2. Requirements

Where discretion is a correctness issue, durability brings performance into the correctness domain. Thus, a program is not durable unless it has access to enough resources to guarantee that it will run; durability requires concurrency so that the process can run, and resource management so that it can guarantee that it will.