Agoric Kernel API Assessment

Security Assessment Report

Agoric

Prepared for Agoric Systems Operating Company
September 21, 2022 (version 1.1)

Project Team:

Technical Testing	Jordan Whitehead and Loren Browman
Technical Editing	Darren Kemp and Joshua Vaughn
Project Management	Sara Bettes

Table of Contents

Engagement Overview

Assessment Components and Objectives

Agoric Systems Operating Company ("Agoric") recently engaged Atredis Partners ("Atredis") to perform a Kernel API Assessment of the Agoric platform. Objectives included validation that the lowest level of the Agoric smart-contract platform was developed and deployed with security best practices in mind.

Testing was performed from April 4, through April 20, 2022, by Jordan Whitehead and Loren Browman of the Atredis Partners team, with Sara Bettes providing project management and delivery oversight. For Atredis Partners' assessment methodology, please see Appendix I of this document, and for team biographies, please see Appendix II. Specific testing components and testing tasks are included below.

COMPONENT	ENGAGEMENT TASKS
Agoric Kernel API Assessment	

Assessment Targets	• Agoric Kernel API
	  • JavaScript-based kernel
	  • Provides interface that Agoric Vats use to communicate with the kernel
	  • Approximately 13 syscalls and supporting functionality
	  • Unit test shim for Vats provides direct kernel interface accessibility

Assessment Tasks	• Source-Assisted Penetration Testing of the Agoric Kernel API
	  • Analyze kernel architecture and threat model
	  • Validate attack surface & define test cases
	  • Manual and automated testing or each kernel API operation
	  • Proof of Concept (PoC) generation, validation, and documentation of findings

Reporting and Analysis	
Analysis and Deliverables	• Status Reporting and Realtime Communication • Comprehensive Engagement Deliverable • Engagement Outbrief and Remediation Review

The ultimate goal of the assessment was to provide a clear picture of risks, vulnerabilities, and exposures as they relate to accepted security best practices, such as those created by the National Institute of Standards and Technology (NIST), or the Center for Internet Security (CIS). Augmenting these, Atredis Partners also draws on its extensive experience in secure development and in testing high-criticality applications and advanced exploitation.

Engagement Tasks

Atredis Partners performed the following tasks, at a high level, for in-scope targets during the engagement.

Runtime Analysis

For relevant software targets identified during the course of this engagement, Atredis performed runtime analysis, using debugging and static analysis tools to analyze application flow to aid in software security analysis. Where relevant, purpose-built tools such as fuzzers and customized clients were utilized to aid in vulnerability identification.

Source Code Analysis

Atredis Partners reviewed the in-scope application source code, with an eye for security-relevant software defects. To aid in vulnerability discovery, application components were mapped out and modeled until a thorough understanding of execution flow, code paths, and application design and architecture were obtained. To aid in this process, the assessment team engaged key stakeholders and members of the development team, where possible, to provide structured walkthroughs and interviews, helping the team rapidly gain an understanding of the application's design and development lifecycle.

Configuration and Architecture Review

Atredis performed a high-level review of available documentation and configuration data with an eye toward the overall functional design and soundness of the implementation. A key aspect of this component was to identify gaps in the architecture and design regarding aspects of design that reduce overall defensibility, aimed at pointing out fundamental issues in the application architecture that should be addressed early in the development cycle as opposed to later when the platform is closer to a full production state.

While specific vulnerabilities may be identified during the architecture and configuration review, the intent was less on finding individual defects and more on how the design of a given target affects its overall defensibility. Outcomes of the architecture review helped to inform testing objectives throughout the rest of the engagement while also helping the client define a long-term platform maturity and security design roadmap.

Executive Summary

Testing targeted the Agoric Kernel's API with the untrusted vats, which is only a segment of the whole Agoric system. Atredis Partners assessed the system as seen in the public `agoric-sdk` repository at the commit beginning with `63d329`. The Agoric team provided instructions for properly building and testing local instances of the system. The Agoric's `swingset-runner` tool was used to quickly establish testing environments targeting the kernel.

Atredis Partners performed testing from the perspective of an attack-controlled vat attacking the kernel, with a focus on verifying the unique responsibilities of the kernel's security boundary. After isolating the security relevant responsibilities of the kernel, Atredis built harnesses and tests to exercise the relevant code paths. Specific auditing was done for the translation and handling of syscalls and deliveries, as these are the primary communication methods between the vats and the kernel.

The focus of the testing was the boundary between the kernel and the vats (a vat being a worker process used for running user defined JavaScript), but this is not the only security boundary within the system. The security boundary between untrusted userspace code and the supervising layers within the vat was not directly tested. The security boundary that protects against malicious remote interactions was also not tested during this review.

Key Conclusions

Overall, Atredis Partners found the architecture of Agoric's kernel to be well designed from a security perspective, properly enforcing the interactions with the vats to ensure proper scoping and access restriction. The architectural design effectively enables enforcing access control through the kernel's reference translation mechanisms.

Further development is needed to add proper vat isolation on the operating system level; Atredis was able to identify key issues here that could lead to subversion of the kernel's security guarantees. Discussion with the Agoric team indicated this is an active area of development and plans already exist to add proper operating system isolation for the vats.

Atredis Partners did not identify any issues that allowed untrusted userspace code to compromise the Agoric Kernel and gain improper access or otherwise compromise the environment. The boundary around userspace was not directly tested, and further testing could be needed here.

As in any security assessment, some general areas for improvement were noted, but overall Atredis Partners would rate the tested components of Agoric's platform as sound from a security perspective and well-aligned with modern secure development practices.

Platform Overview

The tested Agoric platform is designed to enable untrusted JavaScript code to be used deterministically in distributed systems. This JavaScript is limited by using a SES (Secure ECMAScript) runtime which compartmentalizes the untrusted code; restricting its access and seeking to remove mechanisms that could cause any non-deterministic logic. Deterministic execution of the untrusted code is vital, as this system is intended to be able to run smart-contracts, which depend on deterministic replay-ability.

The untrusted code, referred to as "userspace", is supported by a "liveslots" library and a supervisor layer running in the same JavaScript engine. The liveslots component abstracts interactions with the kernel away from userspace and helps enforce deterministic execution. Liveslots must restrict userspace's interactions with the kernel to valid actions. A userspace unrestricted by liveslots could emit invalid syscalls and use garbage collection to enable non-deterministic logic.

Another important job of liveslots is to collect and report accurate metering data. The liveslots component collects this information and reports it to the kernel, which can use it to halt vats, or schedule other vats according to priority.

The SES runtime used is provided by Agoric's JavaScript shim on top of the XS JavaScript engine. Each untrusted vat is run inside a `xsnap-worker` process which runs the liveslots and userspace using the XS engine. This worker process supports the snapshotting of the vat and communication with the kernel.

The kernel of the system is responsible for starting and scheduling the worker processes, and acts as the hub for dispatching messages and item references between vats. The kernel also commits state changes for the vats at appropriate times. The kernel must only allow vats to access objects they have been given references to, and to access state within the vats scope.

Kernel Security Boundary

The testing focused on the security boundary between the kernel and the `xsnap-worker` processes. The Agoric system's architecture depends on multiple different security boundaries, each with unique responsibilities. The security boundary between the kernel and the workers is an important one as it provides security in depth beyond the security boundary provided between userspace and liveslots. Any exploitable vulnerability in liveslots or the XS JavaScript engine could allow an attacker to control the worker process, and directly interact with the kernel.

The Agoric Kernel must provide guarantees that the vats cannot access a reference to an object, promise, or device that was not given to them. To do this the kernel keeps a translation layer between kernel references and references held by each vat. This translation mechanism is used whenever an incoming syscall is parsed from a vat, or whenever an outgoing delivery is crafted for a vat. This acts as an effective barrier by keeping unique translations for each separate vat, and not letting the vat have direct influence over these translations. If a vat were able to reference objects that it had not been given access to, then an attacker-controlled vat could craft references that would allow access to admin devices and expose sensitive objects in other vats.

Another important piece of the kernel's security boundary is the scoping of stored state. The kernel must ensure that each vats committed state cannot be influenced by another vat. The kernel currently does this by prefixing the keys used when storing the data. Keys associated with a vat will have that vat's ID in the prefix, as well as a value indicating what kind of value is being stored. Vat controlled data is stored with the `${vatID}.vs.` prefix, for example. If there were errors in this state scoping, it could allow an attacker-controlled vat to modify other vat's stored values or modify its reference translation table.

Many of the syscalls the kernel handles support garbage collection for items imported or exported by the vats. Accurate garbage collection depends on cooperation with liveslots, but the kernel should ensure that a vat cannot abuse the garbage collection system to break the scoping or access enforcement mentioned above. Fortunately, the garbage collection system does not reuse identifiers that have gone out of scope and is not vulnerable to many traditional attacks associated with reusing freed objects. The architectural design of the garbage collection system means that mistakes should only affect system stability, and not provide attackers an opportunity to exploit the system when references are reused improperly.

An important requirement of the kernel's security boundary is the isolation of each vat. Not only is it important that vats cannot maliciously influence each other to support the two previously mentioned guarantees, but vats should not be allowed to communicate with each other cooperatively without going through the kernel. If vats could communicate without the kernels involvement, they could bypass the imposed ordering and control the kernel uses to help the system stay deterministic and monitored. If not properly isolated, the vats could also use the host system to influence the actions of other vats, breaking the scoping and access requirements imposed by the kernel.

The kernel is also responsible for checking that incoming syscalls from the vats are properly formatted. Invalid syscalls will result in the termination of the offending vat. These checks are important, as mistyped data could lead to unexpected results and possible attacker influence over critical kernel logic. This boundary is also supported by liveslots; usermode code should not have the ability to craft arbitrary syscalls. As a result, validating syscalls is the responsibility of both liveslots and the kernel. To further clarify security roles, Liveslots should be viewed as an extra layer of filtering leaving the kernel to take ownership invaliding malformed syscalls. Care should be taken in any system with multiple security boundaries to make sure each boundaries responsibility is explicit.

When defining the security boundary of the kernel in a system that contains multiple boundaries with separate responsibilities, it is important to also define what guarantees are not being provided. The Agoric Kernel depends on the boundary between liveslots and userspace for many things, including enforcement of determinism. If an attacker has broken the boundary around userspace and has control over the vat process, then the kernel boundary cannot currently ensure that the vat's actions are deterministic. At that point the vat can use timing information or other information from the host system to enable non-deterministic execution. While the kernel provides many mechanisms used by liveslots to maintain a deterministic system, the responsibility of enforcing system determinism can only be on the shoulders of liveslots and XS.

The kernel also cannot guarantee that a compromised vat will not crash the system. While code in userspace should not be able to cause the system to crash or panic, once the vat is fully controlled by an attacker, the kernel may have no better option than to crash at unexpected input or actions. If non-deterministic actions are detected, the kernel has no way to gracefully respond, and should stop execution quickly while leaving logs that can be used to identify the source of the problem.

The kernel can use metering information given by liveslots to schedule or stop vats according to their amount of computation used. But here again the responsibility for ensuring proper metering is up to the liveslots components. A vat that is fully controlled by an attacker can simply lie about their metering usage to the kernel, and proper metering cannot be enforced.

While investigating the security boundary between the vats and the kernel the importance of the security between userspace and liveslots was made even more apparent. The kernel cannot enforce many of the systems requirements alone. A compromised vat could currently cause havoc on the system through careful use of non-deterministic actions and fake metering data. If the boundary between userspace and liveslots is not sufficient, architectural changes could be made to add more responsibility to the kernel's security boundary. Changes such as running each vat in a deterministic emulator could be used to further enforce the system's determinism requirements.

Instrumentation & Analysis

The `agoric-sdk` repository contains a `swingset-runner` package which will run an instance of the kernel with some user defined vats. It contains many helpful demos and examples of tests that will exercise the system with custom code that will run in the userspace of the vats.

For testing the kernel boundary, Atredis Partners needed finer gained control over the communications between the vat and the kernel than `swingset-runner` alone could provide. To accomplish this, a proxy for the `xsnap-worker` binary was created that would detect if a raw vat was being setup. If so, the proxy would launch a separate handler, otherwise it would run the original `xsnap-worker`. A working client was created that could respond how a real `xsnap-worker` hosted vat would respond.

This testing harness was instrumental in validating the kernel's security boundary. By crafting tests that would not be possible from userspace Atredis was able to dynamically validate kernel logic. Agoric could add similar functionality to the existing `swingset-runner` to benefit future dynamic testing of the kernel.

Syscall Summaries

The primary mechanism used by a vat to communicate with the kernel is through the 13 different syscalls available to the vats. These syscalls allow vats to send messages to objects on other vats or devices, inform the kernel on the state of objects for garbage collection, store state in the key store, and work with promises. Below we summarize the security implications considered when auditing each syscall.

send

`send` is the main syscall for interacting with objects exposed by other vats. Methods on an object exported by a vat may be invoked using `send` by specifying the remote object, the method name, and arguments to the remote method.

```
syscall[v3].send(o-53/ko25).talkToBot(@o-50, "encouragementBot")
```

Example `send` call to an object

The target object may be an object or promise that is specified using the slot identifier string. After translation, the slot must map to an existing valid kernel object or promise reference (`koNN` or `kpNN` respectively). Formatted messages are then placed into the kernels acceptance queue to be processed and delivered later.

Specifying a promise as the target of a send can be used to pipeline calls without the sender vat awaiting the resolution of the promise.

```
syscall[v2].send(p+5/kp41).second(@p+5)
```

Example `send` call to a promise

Arguments contained in `send` syscalls destined for remote object methods pass though the kernel to be parsed later by the corresponding remote object which does not a pose a risk to the kernel security boundary. The kernel does however validate the message body contains `CapData` strings and an array of slots.

The `send` syscall presents an opportunity for a compromised vat to send messages to object references it may not have access to which includes references held by other regular vats or privileged vats. Protection against such attacks is enforced during translation and was inspected by Atredis. Vats sending to a target reference not previously allocated by the vat or without a matching c-list entry will assert a failure and terminate the offending vat.

```
vat v1 terminated: error during translation: Error: unknown vatSlot "p-99" ["send","p-
99",{"method":"ping","args":{"body":"[]","slots":[]},"result":"p-60"}]
RAW: Got response from kernel: "error","syscall translation error: prepare to die"]
error during syscall translation: (Error#1)
Error#1: unknown vatSlot p-99
```

Error caught when translating an unknown `vatSlot`

subscribe

`subscribe` is used to register a vat for notification when a promise is resolved. Currently, issuing an eventual send `E()` will automatically trigger a `subscribe` to the generated promise so the calling vat can be notified when the promised is resolved. The `subscribe` syscall requires an identifier to the promise object.

```
syscall[v3].subscribe(p+5/kp41)
```

Example valid `subscribe` syscall

Atredis confirmed promise identifiers are also protected during the slot translation phase from vat space to kernel space. The example below shows the failure when a vat directly specifies a kernel reference (`kref`) instead of a valid vat reference (`vref`).

```
vat v1 terminated: error during translation: Error: invalid vref (a string)
["subscribe","kp1"]
error during syscall translation: (Error#1)
Error#1: invalid vref kp1
```

Example invalid `subscribe kref` syscall

These checks also validate the `vref` exists:

```
vat v1 terminated: error during translation: Error: unknown vatSlot "p-99" ["subscribe","p-
99"]
error during syscall translation: (Error#1)
Error#1: unknown vatSlot p-99
```

Example invalid `subscribe` `vref` syscall

resolve

`resolve` can be used to resolve a promise and may only be called by the decider of the promise. The `resolve` arguments include the promise object to be resolved, the result status, and any promise result data to return to subscribers.

```
syscall[v2].resolve[0](p-60/kp41, false) = "Thanks for the setup. I sure hope I get some
encouragement...\nuser vat is happy\n" []/[]
```

Example valid `resolve` syscall

Aside from the regular checks performed during translation, an additional and important security check is performed to ensure the vat resolving the promise is the decider of that promise. This check is performed in `KernelKeeper.js:623`:

```
assert(
  p.decider === expectedDecider,
  X`${kpid} is decided by ${p.decider}, not ${expectedDecider}`,
);
```

Promise decider validation logic

The promise must be in an `unresolved` state to be resolved. This check is performed in `vatTranslator.js:45`:

```
assert(
    p.state === 'unresolved',
    X`result ${msg.result} already resolved`,
  );
```

Validation of promise state

exit

`exit` is used in situations where a vat wishes to terminate itself. `exit` is passed an `isFailure` flag and some additional `CapData` which are passed to `terminateVat` inside the kernel. `terminateVat` will delete vat state, resolve orphaned promises, notify the parent, and shutdown the worker.

```
syscall[v1].exit(true,{body: "[]", slots: []})
```

Example valid `exit` syscall

The `CapData` argument can be used to provide more information regarding the reason for termination or current task completion status. The `isFailure` flag can be used to avoid committing state in the event of a vat failure. `VATadmin` is then notified the vat has been terminated via a call to `notifyTerminaion`.

Vats cannot abuse the `exit` syscall to terminate other vats as the `vatID` is derived in the kernel and is not a vat supplied parameter.

dropImports, retireImports, retireExports

`dropImports` is part of the distributed garbage collection and will mark all imports specified by the `vrefs` argument as unreachable. This action is performed in `translateDropImports` contained in `vatTranslator.js`. The supplied `vrefs` are converted into `krefs` and marked unreachable by the garbage collector.

```
syscall[v3].dropImports(ko23 ko24 ko25 ko20 ko26)
```

Example `dropImports` syscall

`retireImports` is like `dropImports` except the c-list entry is deleted entirely for the supplied import `vrefs`. This distinction makes retired imports unrecognizable and ready for garbage collection. The supplied `vrefs` must be made unreachable prior to calling `retireImports`.

```
syscall[v3].retireImports(ko23 ko24 ko25 ko20 ko26)
```

Example `retireImports` syscall

`retireExports` is much the same as `retireImports` except the supplied `vrefs` to be deleted from the c-list are exports previously allocated by the calling vat.

All garbage collection related syscalls are protected from releasing references which they do not have access to. This is accomplished by the translation layer and by `mapVatSlotToKernelSlot`. Any specified `vref` is first converted to its c-list equivalent in the form `vN.c.o-NN`. If the any of the c-list entries do not exist for the `vrefs` in the array, the vat will be terminated as demonstrated below.

```
vat v1 terminated: error during translation: Error: vref o-1 not in clist
["dropImports",["o-1","o-49"]]
error during syscall translation: (Error#1)
Error#1: vref o-1 not in clist
```

Invalid `vref` causes vat termination

vatstoreGet, vatstoreSet, vatstoreDelete

These 3 syscalls provide read, write, and delete primitives for accessing the `vatstore`. `vatstore` is additional storage space managed by the kernel and each vat may only access its own data. `vatstore` keys are first translated into a scoped representation with the form `v1.vs.idCounters` where `vN` specifies the vat and vs denotes a `vatstore` value. Values are accessed using the underlying `kvStore.get`, `kvStore.set` and `kvStore.delete` functions after calling `vatstoreKeyKey` to add the unique prefix.

```
syscall[v1].vatstoreGet(idCounters)
syscall[v3].vatstoreSet(vc.1.|entryCount,0)
syscall[v3].vatstoreDelete(lp20.status)
```

Example `vatstore` syscalls

These `vatstore` primitives provide direct access from user space to the kernel managed storage. The default storage is a LMDB persistent storage. It is possible to exceed hard-coded LMDB resource limits and crash the kernel as demonstrated in Exceeding LMDB Map Size Limit Causes Kernel Crash and Unvalidated vatstore Key Length Causes Kernel Crash.

Other attack vectors include modifying sensitive internal state and `vatstore` values owned by other vats. Atredis analyzed how `vatstore` key strings are built in the kernel and determined they do not allow attackers to perform actions to modify values they should not have access to.

vatstoreGetAfter

`vatstoreGetAfter` allows the caller to iterate over keys in its `vatstore`. `vatstoreGetAfter` takes several arguments including the upper and lower bounds for iteration range. A call to `vatstoreGetAfter` only executes one step and can be called successively in a loop until undefined is returned.

```
syscall[v3].vatstoreGetAfter(, vom.kind., undefined)
```

Example `vatstoreGetAfter` syscall

`vatstoreGetAfter` accepts UTF-16 encoded characters for all arguments. This alone does not pose a security risk but was analyzed to confirm it did not trigger any odd behaviour.

Atredis verified access is restricted to the owning vat and only values contained in the `vatstore` with the correct key prefix `vN.vs` were returned. The prefix is not user-controlled input and is prepended by the kernel when `vatstoreKeyKey` is called.

```
const actualPriorKey = vatstoreKeyKey(vatID, priorKey);
const actualLowerBound = vatstoreKeyKey(vatID, lowerBound);
```

Prepending safe `vatID` in `kernelSyscall.js:150`

callNow

`callNow` is similar to `send` but is synchronous in nature and will not accept a promise as an argument. `callNow` is intended to enable an immediate interface to device nodes.

```
syscall[v7].callNow(d-70/kd32).add({"body":"[\"bot\",1,\"1:0:deliver:ro+0:encourageMe:rp-40;[\\\"user\\\"]\"]","slots":[]})
```

Example `callNow` syscall

`callNow` is also protected by the translation layer in that a vat cannot send messages to devices which do not have a corresponding entry in the vat's c-list.

```
vat v1 terminated: error during translation: Error: unknown vatSlot "d-1" ["callNow","d-1","blah",{"body":"[]","slots":[]}]
error during syscall translation: (Error#1)
Error#1: unknown vatSlot d-1
```

Vat termination when specifying an unknown vat

Additional checks also validate the target is in fact a device and not an object or a promise.

```
const { type } = parseKernelSlot(dev);
assert(type === 'device', X`doCallNow must target a device, not ${dev}`);
for (const slot of args.slots) {
  assert(
    parseVatSlot(slot).type !== 'promise',
    `syscall.callNow() args cannot include promises like ${slot}`,
  );
}
```

Device validation logic in `vatTranslator:472`

Delivery Summaries

When the kernel has information for the vats, it uses one of eight deliveries. These deliveries are used to start or stop the vat, inform about the state of objects for garbage collection, and deliver messages and resolved promise data.

While vats cannot directly craft deliveries, the logic behind these communication mechanisms still should be audited for their security impact, as well as the handling of the vat's response. Below we summarize the security implications considered when auditing each delivery.

message

The `message` delivery is sent to a vat targeting a specific object exported by that vat. It is often the result of a `send` syscall. It usually contains a method to be invoked on that object, as well as arguments. The translation of the argument's `CapData` will give the vat access to any objects included in the arguments.

If the `message` were sent to the wrong vat, or if by some other means a vat were given access to items it should not logically be able to reach, then that would be a security issue.

notify

The `notify` delivery is sent to any vat subscribed to a promise that has been resolved. This delivery contains an argument with the data the related promises resolved to, or associated error data. The kernel will recursively walk the promise data searching for all included promise data.

If the kernel could be trapped in an infinite loop while resolving promises it would be an issue. This delivery will give the destination vat access to any new items in the resolution data. As such if it could accidentally include items that should not logically be accessible by the destination vat that would be a security relevant issue.

dropExports, retireExports, retireImports

These three deliveries are used to notify the vats of garbage collection state for shared items. Each has an array argument with references being retired or dropped. These deliveries are only supposed to contain items that the target vat already has access to.

If a logic error allowed these deliveries contain an item that the target vat did not already have access to, that would be a security issue.

startVat

The `startVat` delivery is sent to each vat to give them a chance to initialize and export objects. This delivery contains a `CapData` argument that will give the target vat access to any included items.

It would be a security issue if an attacker were able to cause a `startVat` to be delivered to an attacker-controlled vat with references to items that the attacker cannot already access.

stopVat

`stopVat` is delivered without any arguments and is meant to alert the vat that it is being stopped. Because this delivery cannot provide any further access or control to a vat, it does not raise any obvious security concerns.

bringOutYourDead

This delivery is sent to the vats after a certain number of deliveries, and has no arguments. This is meant to prompt the vats to issue garbage collection syscalls so that the kernel can clean up left over items.

Because this delivery does not grant any access or information to the vat it does not raise any obvious security concerns.

Other Communication Mechanisms

Syscalls and deliveries are not the only type of message passed between an `xsnap-worker` and the kernel. The kernel can emit many types of messages to control the vat including telling it to import files, evaluate JavaScript, use a packaged bundle of code, and write a snapshot. For any of these commands that are unidirectional from the kernel to the vat, the relevant security consideration is in how the kernel handles the vat's response. Fortunately, the responses are typically not complex and simple to handle.

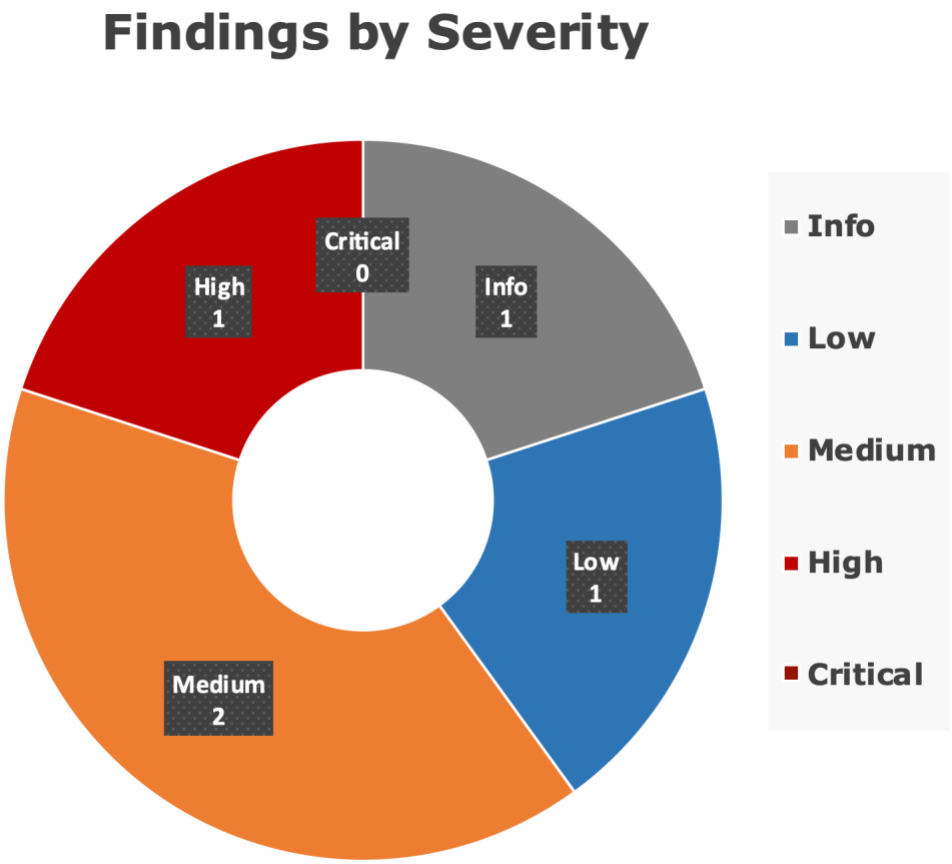
Syscalls are one type of query a `xsnap-worker` hosted vat can send to the kernel, but not the only one. The other types of queries currently involve logging messages. These provide a mechanism the vats can use to add items to the log, and the kernel will filter and annotate the messages accordingly. The security considerations with these other mechanisms have to do with ensuring that logs can be trusted, and not allowing spoofed log entries to be created by a vat pretending to speak for another vat or the kernel.

Findings Summary

In performing testing for this assessment, Atredis Partners identified **one (1) high**, **two (2) medium**, **one (1) low** severity findings, and **one (1) informational** finding. No critical severity findings were noted. As stated earlier, none of these issues constitute a potential for direct compromise from userspace, and in the case of the high severity vulnerability, the Agoric team has already noted development plans to address the issue.

Atredis defines vulnerability severity ranking as follows:

- **Critical:** These vulnerabilities expose systems and applications to immediate threat of compromise by a dedicated or opportunistic attacker.
- **High:** These vulnerabilities entail greater effort for attackers to exploit and may result in successful network compromise within a relatively short time.
- **Medium:** These vulnerabilities may not lead to network compromise but could be leveraged by attackers to attack other systems or applications components or be chained together with multiple medium findings to constitute a successful compromise.
- **Low:** These vulnerabilities are largely concerned with improper disclosure of information and should be resolved. They may provide attackers with important information that could lead to additional attack vectors or lower the level of effort necessary to exploit a system.


Findings by Severity
Critical
0
Info
1
High
1
Low
1
Medium
2
Info
Low
Medium
High
Critical

Findings and Recommendations

The following section outlines findings identified via manual and automated testing over the course of this engagement. Where necessary, specific artifacts to validate or replicate issues are included, as well as Atredis Partners' views on finding severity and recommended remediation.

Findings Summary

The below tables summarize the number and severity of the unique issues identified throughout the engagement.

CRITICAL	HIGH	MEDIUM	LOW	INFO
0	1	2	1	1

Findings Detail

FINDING NAME	SEVERITY
Vats Lack Isolation	**High**
Exceeding LMDB Map Size Limit Causes Kernel Crash	**Medium**
Unvalidated vatstore Key Length Causes Kernel Crash	**Medium**
Log Injection via Standard Output	**Low**
Crash When Sending to Device	**Info**

Vats Lack Isolation

Severity: High

Finding Overview

The Agoric Kernel does not enforce any kind of host supported isolation on the vat processes. An attacker-controlled vat can take over the communication channels between the kernel and other vats, exposing private resources and violating trust.

Finding Detail

The Agoric architecture uses a `xsnap-worker` when running individual vats. This will run the untrusted usermode code in a new process and under the protections given by the XS JavaScript engine.

To provide defense in depth in the event the XS engine or liveslots cannot be trusted, the kernel provides a security boundary against compromised vats. However, the `xsnap-worker` used to start the vats does not provide any additional isolation to contain an untrusted vat.

Because of this it is possible for an attacker-controlled vat to access the file system, kill other processes, and other dangerous actions. It is also possible for attackers to gain access to the data streams between the kernel and its vats.

When the `xsnap-worker` processes are created, they inherit file descriptors `3` and `4`. Nodejs on Linux implements these data streams with unnamed Unix sockets connected between the vat and the kernel.

```
$ ls -l /proc/3483/fd
total 0
lr-x------. 1 vm vm 64 Apr 13 13:15 0 -> /dev/null
l-wx------. 1 vm vm 64 Apr 13 13:15 1 -> 'pipe:[20267]'
l-wx------. 1 vm vm 64 Apr 13 13:15 2 -> 'pipe:[20267]'
lrwx------. 1 vm vm 64 Apr 13 13:15 3 -> 'socket:[44361]'
lrwx------. 1 vm vm 64 Apr 13 13:15 4 -> 'socket:[44363]'
```

The file descriptors for a vat showing `3` and `4` are sockets back to the Agoric Kernel

An attacker-controlled vat can use the Linux system call `pidfd_getfd` to duplicate important descriptors from the kernel or other vats into their own process. With access to these private data streams, the malicious vat can spoof messages from the kernel to the other vats, or from the other vats to the kernel.

Note that `pidfd_getfd` is only supported since Linux 5.6 and requires the process to pass a `PTRACE_MODE_ATTACH_REALCREDS` check, which is the same check required for attaching a debugger to a process. By default, on many distributions this security check will pass for a process targeting another process owned by the same user. As the kernel and vat processes all run as the same user, no special permissions changes were needed when writing a proof of concept.

```
XSNAP_DBG: Sending 3110(Malicious-vat) a query:
["deliver",["message","o+0",{"method":"stealpipes","args":{"body":"[{\"@qclass\":\"slot\",\
"iface\":\"Alleged: root\",\"index\":0}]","slots":["o-50"]},"result":"p-60"}]]
Writing to stolen pipes
Executing /home/vm/agoric-kernel-2022/compromized_vat/handler/fdstealer
Stealing fd 4 from 3121(target-vat)
injecting a spoofed vatstoreSet from the target vat
fdstealer done

/* ... Later when a message is delivered to the target vat ... */

XSNAP_DBG: Sending 3121(target-vat) a query:
["deliver",["message","o+0",{"method":"ping","args":{"body":"[]","slots":[]},"result":"p-
60"}]]
XSNAP_DBG: 3121(target-vat) sent
?["syscall",["vatstoreSet","PrivateStoreKey","MaliciousData"]]
```

Output from an instrumented kernel and vat while target vat is made to send a malicious syscall

A malicious vat with access to these data streams can issue syscalls from other vats or deliver messages to the other vats without requiring a valid reference to the target objects.

Recommendation(s)

The Agoric team was already aware of a need for further isolation of the vats, and they have plans to isolate the vats with **"**secure computing**"** (seccomp) or a similar mechanism. If properly implemented this could successfully be used to prevent an attacker-controlled vat from using the operating system to undermine the kernel.

Configuration of the system with a Linux Security Module (LSM) such as Security-Enhanced Linux (SELinux) or AppArmor could also be used to limit the actions the vats can perform.

Fixes for this issue should not simply suppress the `pidfd_getfd` mechanism, but rather seek to isolate the vat from doing any unnecessary interaction with the host operating system.

References

CWE-653: Improper Isolation or Compartmentalization:
https://cwe.mitre.org/data/definitions/653.html

Seccomp BPF documentation:
https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html

Issue in Agoric's repository discussing vat isolation:
https://github.com/Agoric/agoric-sdk/issues/2386

Exceeding LMDB Map Size Limit Causes Kernel Crash

Severity: Medium

Finding Overview

The Lightning Memory-Mapped Database (LMDB) map size is not monitored when pushing `vatstore` values to persistent storage. As a result, the kernel crashes when committing values to the `vatstore` without causing a kernel panic.

Finding Detail

The maximum size of the LMDB database is defined in swingStore.js:

```
export const DEFAULT_LMDB_MAP_SIZE = 2 * 1024 * 1024 * 1024;
```

LMDB size set to 2GB in `swingStore.js`

The following proof of concept was created to test if code running in a vat may exceed this defined limit:

```
import { E } from '@endo/eventual-send';
import { Far } from '@endo/marshal';

export function buildRootObject(vats) {
  return Far('root', {
    bootstrap(vats) {
      let maxcalls = 200;
      for (let i = 0; i < maxcalls; i++) {
        E(vats.bob).doStuff(String(i));
      }
    }
  });
}
```

Relevant section in `bootstrap.js`

```
import { Far } from '@endo/marshal';

export function buildRootObject(vatPowers) {
  let c1 = VatData.makeScalarBigMapStore('myData');
  let chunk = 10 * 1024 * 1024; //10MB chunks

  return Far('root', {
    doStuff(name) {
      console.log(`=> writing entry: ${name}, total bytes: ${parseInt(name)*chunk}`);
      c1.init(name, 'B'.repeat(chunk))
    }
  });
}
```

Relevant section in `fvat-bob.js`

When executing this test, the LMDB map size is eventually exceeded causing the kernel to crash without causing a kernel panic.

```
=> writing entry: 100, total bytes: 1048576000
=> writing entry: 101, total bytes: 1059061760
kernel failure in crank 528: Error: MDB_MAP_FULL: Environment mapsize limit reached
(Error#1)
Error#1: MDB_MAP_FULL: Environment mapsize limit reached

  at Txn.putString (<anonymous>)
  at Object.set (packages/swing-store/src/swingStore.js:244:9)
  at Object.commitCrank (.../swingset-vat/src/kernel/state/storageWrapper.js:182:15)
  at processDeliveryMessage (.../swingset-vat/src/kernel/kernel.js:1073:54)
  at async Object.step (.../swingset-vat/src/kernel/kernel.js:1455:7)
  at async runBlock (packages/swingset-runner/src/main.js:605:25)
  at async runBatch (packages/swingset-runner/src/main.js:673:15)
  at async commandRun (packages/swingset-runner/src/main.js:693:32)
  at async main (packages/swingset-runner/src/main.js:452:7)
```

LMDB map size exceeded error

Recommendation(s)

Monitor the current database usage and terminate any vat wishing to exceed the limits of the current LMDB size or cause a kernel panic.

References

CWE-400: Uncontrolled Resource Consumption:
https://cwe.mitre.org/data/definitions/400.html

LMDB Error:
https://github.com/LMDB/lmdb/blob/4b6154340c27d03592b8824646a3bc4eb7ab61f5/libraries/liblmdb/mdb.c#L1694

Unvalidated vatstore Key Length Causes Kernel Crash

Severity: Medium

Finding Overview

The key size set when initializing a `vatstore` value is not validated against the maximum allowable key size for the underlying LMDB persistent storage. As a result, the kernel crashes when committing values to the `vatstore` without causing a kernel panic.

Finding Detail

Creating a key with string length value greater than 242 bytes causes the kernel to crash from userspace. The following example demonstrates the issue.

```
export function buildRootObject(_vatPowers) {
    let c1;

    return Far('root', {
      doStuff(name) {
        console.log('=> Bob: doing Stuff! ----------------------------------------');

        c1 = VatData.makeScalarBigMapStore('myData');
        c1.init('A'.repeat(243), 'B'.repeat(32));
      }
    });
  }
```

Code running in a vat to generate large key value

Running this code causes triggers the following LMDB error and resulting kernel crash.

```
node bin/runner --verbose --usexs --init run demo/vatStoreKeyLength
[SNIPPED]
=> Bob: doing Stuff! -------------------------------------------
syscall[v2].vatstoreSet(vc.2.|nextOrdinal,1)
syscall[v2].vatstoreSet(vc.2.|entryCount,0)
syscall[v2].vatstoreSet(vc.2.|schemata,{"body":"[{\"@qclass\":\"tagged\",\"tag\":\"match:sc
alar\",\"payload\":{\"@qclass\":\"undefined\"}}]","slots":[]})
syscall[v2].vatstoreSet(vc.2.|label,myData)
syscall[v2].vatstoreGet(vc.2.sAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
)
syscall[v2].vatstoreSet(vc.2.sAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
,{"body":"\"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB\"","slots":[]})
syscall[v2].vatstoreGet(vc.2.|entryCount)
syscall[v2].vatstoreSet(vc.2.|entryCount,1)
syscall[v2].resolve[0](p-60/kp41, false) = {"@qclass":"undefined"} []/[]
Delete mapping v2.c.kp41<=>v2.c.p-60
syscall[v2].vatstoreSet(idCounters,{"exportID":10,"collectionID":3,"promiseID":5})
kernel failure in crank 34: Error: MDB_BAD_VALSIZE: Unsupported size of key/DB name/data,
or wrong DUPFIXED size (Error#1)
Error#1: MDB_BAD_VALSIZE: Unsupported size of key/DB name/data, or wrong DUPFIXED size

  at Object.set (packages/swing-store/src/swingStore.js:244:9)
  at Object.commitCrank (.../swingset-vat/src/kernel/state/storageWrapper.js:182:15)
  at processDeliveryMessage (.../swingset-vat/src/kernel/kernel.js:1073:54)
  at async Object.step (.../swingset-vat/src/kernel/kernel.js:1455:7)
  at async runBlock (packages/swingset-runner/src/main.js:605:25)
  at async runBatch (packages/swingset-runner/src/main.js:673:15)
  at async commandRun (packages/swingset-runner/src/main.js:693:32)
  at async main (packages/swingset-runner/src/main.js:452:7)
```

LMDB error and kernel crash

No kernel panic or vat termination was noted in the above error output. The LMDB key length limit appears to be 248 bytes total including header bytes of the form `vc.2.s` consuming 6 bytes, this allows for a maximum key size of 242 before causing the error.

Atredis also validated that any messages destined to other vats to be executed on future cranks are not executed.

Recommendation(s)

The key length for all `vatstore` values should be validated to ensure the length does not exceed what is allowed by LMDB. This condition could result in vat termination, or a kernel panic as opposed to an uncaught kernel error.

References

CWE-20: Improper Input Validation:
https://cwe.mitre.org/data/definitions/20.html

LMDB Error:
https://github.com/LMDB/lmdb/blob/4b6154340c27d03592b8824646a3bc4eb7ab61f5/libraries/liblmdb/mdb.c#L1705

Log Injection via Standard Output

Severity: Low

Finding Overview

A vat has access to the same `stdout` and `stderr` files that the kernel uses. An attacker-controlled vat can use this to spoof log messages, bypassing the intended `console` path. This vulnerability could let an attacker produce confusing or misleading log files.

Finding Detail

Usermode code running in an `xsnap-worker` vat will have its calls to `console.log` redirected through the kernel, using a special `console` query message.

```
?["console","log","My debug message"]
```

An example console query sent to the kernel from a vat

By redirecting this output through the kernel, the system has to ability ignore certain messages, restrict, and redirect vat output, and prefix all messages with the vat's identifier.

```
case 'liveSlotsConsole':
case 'console': {
    const [level, ...rest] = args;
    // Choose the right console.
    const myConsole =
    (type === 'liveSlotsConsole' && liveSlotsConsole) || vatConsole;
    if (typeof level === 'string' && level in myConsole) {
    myConsole[level](...rest);
    } else {
    console.error(`bad ${type} level`, level);
    }
    return ['ok'];
}
case 'testLog':
    testLog(...args);
    return ['OK'];
```

Code within `manager-subprocess-xsnap.js` that handles the `console` command

Unfortunately, the process that contains the untrusted code itself has access to the same standard output and standard error files that the kernel uses. When the process is spawned it is given access to these files.

```
const xsnapOpts = {
    os: osType(),
    spawn,
    stdout: 'inherit',
    stderr: 'inherit',
    debug: !!env.XSNAP_DEBUG,
  };
```

The options in `controller.js` share the host's `stdout` and `stderr`

If an untrusted vat is compromised by an attacker, the attacker can now use the standard file streams to output messages that are not properly sorted and prefixed as they would be by the `console` command. This could lead to malicious messages in the log that obfuscate or confuse the true actions of the system.

Recommendation(s)

The `stdout` and `stderr` should not be passed to the vats. Instead, these files could be ignored, or they could be pipes that allow the kernel to properly filter and process the vats output.

References

CWE-117: Improper Output Neutralization for Logs:
https://cwe.mitre.org/data/definitions/117.html

Crash When Sending to Device

Severity: Info

Finding Overview

When validating the arguments of a `send` syscall the kernel does not check that the target is a valid type. If a vat does a `send` to a device the kernel will fail an assert and crash when routing the message. This error could possibly be better handled during the validation of the arguments.

Finding Detail

When the Agoric kernel translates a `send` syscall, the target reference is translated to a kernel reference, but the type of this reference is not verified. When the syscall is dispatched, the kernel will assert that the target references a promise or an object. This assert will fail when the target references a device and will throw an uncaught error that will crash the kernel.

```
function routeSendEvent(message) {
    const { target, msg } = message;
    const { type } = parseKernelSlot(target);
    assert(
      ['object', 'promise'].includes(type),
      X`unable to send() to slot.type ${type}`,
    );
    /*...*/
  }
```

The assert in `routeSendEvent` that asserts the target is an `object` or a `promise`

This crash may not be able to be caused from userspace, as imported devices are treated differently from imported objects. As such this is an informational finding, and not currently a security vulnerability.

```
kernel failure in crank 36: Error: unable to send() to slot.type (a string) (Error#1)
Error#1: unable to send() to slot.type device

  at routeSendEvent (.../swingset-vat/src/kernel/kernel.js:756:5)
  at deliverRunQueueEvent (.../swingset-vat/src/kernel/kernel.js:894:21)
  at processDeliveryMessage (.../swingset-vat/src/kernel/kernel.js:956:26)
  at tryProcessDeliveryMessage (.../swingset-vat/src/kernel/kernel.js:1091:12)
  at startProcessingNextMessageIfAny (.../swingset-vat/src/kernel/kernel.js:1440:25)
  at Object.step (.../swingset-vat/src/kernel/kernel.js:1453:31)
  at Object.step (packages/SwingSet/src/controller/controller.js:371:21)
  at runBlock (packages/swingset-runner/src/main.js:605:42)
  at async runBatch (packages/swingset-runner/src/main.js:673:15)
  at async commandRun (packages/swingset-runner/src/main.js:693:32)
  at async main (packages/swingset-runner/src/main.js:452:7)
```

Error message when failing the assert

Recommendation(s)

The translation of `send` syscalls from vats should require that the target is of the correct type before adding the message to the queue.

References

CWE-20: Improper Input Validation:
https://cwe.mitre.org/data/definitions/20.html

Appendix I: Assessment Methodology

Atredis Partners draws on our extensive experience in penetration testing, reverse engineering, hardware/software exploitation, and embedded systems design to tailor each assessment to the specific targets, attacker profile, and threat scenarios relevant to our client's business drivers and agreed upon rules of engagement.

Where applicable, we also draw on and reference specific industry best practices, regulations, and principles of sound systems and software design to help our clients improve their products while simultaneously making them more stable and secure.

Our team takes guidance from industry-wide standards and practices such as the National Institute of Standards and Technology's (NIST) Special Publications, the Open Web Application Security Project (OWASP), and the Center for Internet Security (CIS).

Throughout the engagement, we communicate findings as they are identified and validated, and schedule ongoing engagement meetings and touchpoints, keeping our process open and transparent and working closely with our clients to focus testing efforts where they provide the most value.

In most engagements, our primary focus is on creating purpose-built test suites and toolchains to evaluate the target, but we do utilize off-the-shelf tools where applicable as well, both for general patch audit and best practice validation as well as to ensure a comprehensive and consistent baseline is obtained.

Research and Profiling Phase

Our research-driven approach to testing begins with a detailed examination of the target, where we model the behavior of the application, network, and software components in their default state. We map out hosts and network services, patch levels, and application versions. We frequently use a number of private and public data sources to collect Open Source Intelligence about the target, and collaborate with client personnel to further inform our testing objectives.

For network and web application assessments, we perform network and host discovery as well as map out all available application interfaces and inputs. For hardware assessments, we study the design and implementation, down to a circuit-debugging level. In reviewing source code or compiled application code, we map out application flow and call trees and develop a solid working understand of how the application behaves, thus helping focus our validation and testing efforts on areas where vulnerabilities might have the highest impact to the application's security or integrity.

Analysis and Instrumentation Phase

Once we have developed a thorough understanding of the target, we use a number of specialized and custom-developed tools to perform vulnerability discovery as well as binary, protocol, and runtime analysis, frequently creating engagement-specific software tools which we share with our clients at the close of any engagement.

We identify and implement means to monitor and instrument the behavior of the target, utilizing debugging, decompilation and runtime analysis, as well as making use of memory and filesystem

forensics analysis to create a comprehensive attack modeling testbed. Where they exist, we also use common off-the-shelf, open-source and any extant vendor-proprietary tools to aid in testing and evaluation.

Validation and Attack Phase

Using our understanding of the target, our team creates a series of highly-specific attack and fault injection test cases and scenarios. Our selection of test cases and testing viewpoints are based on our understanding of which approaches are most relevant to the target and will gain results in the most efficient manner, and built in collaboration with our client during the engagement.

Once our test cases are validated and specific attacks are confirmed, we create proof-of-concept artifacts and pursue confirmed attacks to identify extent of potential damage, risk to the environment, and reliability of each attack scenario. We also gather all the necessary data to confirm vulnerabilities identified and work to identify and document specific root causes and all relevant instances in software, hardware, or firmware where a given issue exists.

Education and Evidentiary Phase

At the conclusion of active testing, our team gathers all raw data, relevant custom toolchains, and applicable testing artifacts, parses and normalizes these results, and presents an initial findings brief to our clients, so that remediation can begin while a more formal document is created. Additionally, our team shares confirmed high-risk findings throughout the engagement so that our clients may begin to address any critical issues as soon as they are identified.

After the outbrief and initial findings review, we develop a detailed research deliverable report that provides not only our findings and recommendations but also an open and transparent narrative about our testing process, observations and specific challenges in developing attacks against our targets, from the real world perspective of a skilled, motivated attacker.

Automation and Off-The-Shelf Tools

Where applicable or useful, our team does utilize licensed and open-source software to aid us throughout the evaluation process. These tools and their output are considered secondary to manual human analysis, but nonetheless provide a valuable secondary source of data, after careful validation and reduction of false positives.

For runtime analysis and debugging, we rely extensively on Hopper, IDA Pro and Hex-Rays, as well as platform-specific runtime debuggers, and develop fuzzing, memory analysis, and other testing tools primarily in Ruby and Python.

In source auditing, we typically work in Visual Studio, Xcode and Eclipse IDE, as well as other markup tools. For automated source code analysis we will typically use the most appropriate toolchain for the target, unless client preference dictates another tool.

Network discovery and exploitation make use of Nessus, Metasploit, and other open-source scanning tools, again deferring to client preference where applicable. Web application runtime analysis relies extensively on the Burp Suite, Fuzzer and Scanner, as well as purpose-built automation tools built in Go, Ruby and Python.

Engagement Deliverables

Atredis Partners deliverables include a detailed overview of testing steps and testing dates, as well as our understanding of the specific risk profile developed from performing the objectives of the given engagement.

In the engagement summary we focus on "big picture" recommendations and a high-level overview of shared attributes of vulnerabilities identified and organizational-level recommendations that might address these findings.

In the findings section of the document, we provide detailed information about vulnerabilities identified, provide relevant steps and proof-of-concept code to replicate these findings, and our recommended approach to remediate the issues, developing these recommendations collaboratively with our clients before finalization of the document.

Our team typically makes use of both DREAD and NIST CVE for risk scoring and naming, but as part of our charter as a client-driven and collaborative consultancy, we can vary our scoring model to a given client's preferred risk model, and in many cases will create our findings using the client's internal findings templates, if requested.

Sample deliverables can be provided upon request, but due to the highly specific and confidential nature of Atredis Partners' work, these deliverables will be heavily sanitized, and give only a very general sense of the document structure.

Appendix II: Engagement Team Biographies

Shawn Moyer, Founding Partner and CEO

Shawn Moyer scopes, plans, and coordinates security research and consulting projects for the Atredis Partners team, including reverse engineering, binary analysis, advanced penetration testing, and private vulnerability research. As CEO, Shawn works with the Atredis leadership team to build and grow the Atredis culture, making Atredis Partners a home for some of the best minds in information security, and ensuring Atredis continues to deliver research and consulting services that exceed our client's expectations.

Experience

Shawn brings over 25 years of experience in information security, with an extensive background in penetration testing, advanced security research including extensive work in mobile and Smart Grid security, as well as advanced threat modeling and embedded reverse engineering.

Shawn has served as a team lead and consultant in enterprise security for numerous large initiatives in the financial sector and the federal government, including IBM Internet Security Systems' X-Force, MasterCard, a large Federal agency, and Wells Fargo Securities, all focusing on emerging network and application attacks and defenses.

In 2010, Shawn created Accuvant Labs' Applied Research practice, delivering advanced research-driven consulting to numerous clients on mobile platforms, critical infrastructure, medical devices and countless other targets, growing the practice 1800% in its first year.

Prior to Accuvant, Shawn helped develop FishNet Security's penetration testing team as a principal security consultant, growing red team offerings and advanced penetration testing services, while being twice selected as a consulting MVP.

Key Accomplishments

Shawn has written on emerging threats and other topics for Information Security Magazine and ZDNet, and his research has been featured in the Washington Post, BusinessWeek, NPR and the New York Times. Shawn is a twelve-time speaker at the Black Hat Briefings and has been an invited speaker at other notable security conferences around the world.

Shawn is likely best known for delivering the first public research on social network security, pointing out much of the threat landscape still exists on social network platforms today. Shawn also co-authored an analysis of the state of the art in web browser exploit mitigation, creating the first in-depth comparison of browser security models along with Dr. Charlie Miller, Chris Valasek, Ryan Smith, Joshua Drake, and Paul Mehta.

Shawn studied Computer and Network Information Systems at Missouri University and the University of Louisiana at Lafayette, holds numerous information security certifications, and has been a frequent presenter at national and international security industry conferences.

Loren Browman, Senior Research Consultant

Loren Browman has over 10 years of experience in both consulting and federal law enforcement environments. His experiences range from deep security research in federal government to product and application testing for Fortune 500 corporations. Loren is a recognized subject matter expert (SME) in securing IoT products and advanced hardware testing methodology. Areas of expertise include reverse engineering of hardware, firmware, and communication protocols.

Experience

Loren has conducted numerous large scale product security assessments including challenging black box security assessments and secure design reviews.

Prior to joining Atredis, Loren was an operations supervisor and security researcher for the Royal Canadian Mounted Police (RCMP). This role included providing technical expertise to support police investigations and leading security research efforts in order to circumvent security mechanisms and develop deployable capabilities.

Key Accomplishments

Loren has developed numerous tools for accelerating research on a wide range of products. This includes the development of a fuzzing suite for automotive Electronic Control Units over CAN bus vehicle networks, this led to the discovery of multiple hidden services and exploits. More recently, Loren published nrfsec, a tool for automating firmware recovery vulnerability on secured nrf51 System on Chips.

Loren has studied Electrical and Computer Engineering at the British Columbia Institute of Technology and has attended various specialized training sessions including the Arm IoT Exploit Laboratory, Power Analysis and Glitching and is an Offensive Security Certified Professional (OSCP).

Jordan Whitehead, Senior Research Consultant

Jordan Whitehead specializes in vulnerability research and binary exploitation. Jordan is able to quickly dive into large systems and find key weaknesses as a result of his significant experience in operating system internals.

Experience

During Jordan's Computer Engineering degree schooling, he created and instructed collegiate courses and clubs on computer security. After college he worked as a CNO developer for ManTech International, developing tools and capabilities that involved deep exploration into modern operating systems for exploitable weaknesses. While in that position, Jordan also continued to help create and instruct a number of formal reverse engineering and exploitation courses. These courses detailed the system internals for Windows, Linux, and Android. He has worked with research teams developing custom virtualization and emulation tooling that enabled researchers to better assess otherwise unreachable systems.

Key Accomplishments

Jordan has helped publish papers at top academic conferences on computer security, including Usenix Security Symposium. He has also developed open-source tools related to vulnerability research and secure software. These include peer-reviewed tools that have helped provide useable security and trust on Linux and Windows platforms.

Sara Bettes, Client Operations Associate

Sara Bettes assists the creation and completion of projects at Atredis Partners, ranging from the full pre-sales process to project design and management, to final delivery and follow-up. Her goals are to ensure all projects are executed in a way that reaches the goals of the client and assists the consultants at every turn.

Experience

Prior to joining Atredis Partners, Sara led a team that planned international sporting competitions, Olympic and national team qualifying events, as well as supported the mission of multiple non-profits. Her experience includes Live Sports Commentating, Staffing Management, Safety Plan Creation, Event Development, Public Relations, and Marketing efforts.

Key Accomplishments

Sara earned a bachelor's degree in Mass Communications with an emphasis in Broadcast and Public Relations from Oklahoma City University.

Appendix III: About Atredis Partners

Atredis Partners was created in 2013 by a team of security industry veterans who wanted to prioritize offering quality and client needs over the pressure to grow rapidly at the expense of delivery and execution. We wanted to build something better, for the long haul.

In six years, Atredis Partners has doubled in size annually, and has been named three times to the Saint Louis Business Journal's "Fifty Fastest Growing Companies" and "Ten Fastest Growing Tech Companies". Consecutively for the past three years, Atredis Partners has been listed on the Inc. 5,000 list of fastest growing private companies in the United States.

The Atredis team is made up of some of the greatest minds in Information Security research and penetration testing, and we've built our business on a reputation for delivering deeper, more advanced assessments than any other firm in our industry.

Atredis Partners team members have presented research over forty times at the BlackHat Briefings conference in Europe, Japan, and the United States, as well as many other notable security conferences, including RSA, ShmooCon, DerbyCon, BSides, and PacSec/CanSec. Most of our team hold one or more advanced degrees in Computer Science or engineering, as well as many other industry certifications and designations. Atredis team members have authored several books, including *The Android Hacker's Handbook*, *The iOS Hacker's Handbook*, *Wicked Cool Shell Scripts*, *Gray Hat C#*, and *Black Hat Go*.

While our client base is by definition confidential and we often operate under strict nondisclosure agreements, Atredis Partners has delivered notable public security research on improving the security at Google, Microsoft, The Linux Foundation, Motorola, Samsung and HTC products, and were the first security research firm to be named in Qualcomm's Product Security Hall of Fame. We've received four research grants from the Defense Advanced Research Project Agency (DARPA), participated in research for the CNCF (Cloud Native Computing Foundation) to advance the security of Kubernetes, worked with OSTIF (The Open Source Technology Improvement Fund) and The Linux Foundation on the Core Infrastructure Initiative to improve the security and safety of the Linux Kernel, and have identified entirely new classes of vulnerabilities in hardware, software, and the infrastructure of the World Wide Web.

In 2015, we expanded our services portfolio to include a wide range of advanced risk and security program management consulting, expanding our services reach to extend from the technical trenches into the boardroom. The Atredis Risk and Advisory team has extensive experience building mature security programs, performing risk and readiness assessments, and serving as trusted partners to our clients to ensure the right people are making informed decisions about risk and risk management.