Research Report: Mitigating LangSec Problems With Capabilities

Nathaniel Wesley Filardo
Johns Hopkins University
Baltimore, MD
nwf@cs.jhu.edu

Abstract—**Security and privacy of computation, and the related concept of (deliberate) sharing, have, historically, largely been afterthoughts. In a traditional multi-user, multi-application web hosting environment, typically applications are public by default. Applications wishing to offer a notion of private resources must take it upon themselves to independently manage authentication and authorization of users, leading to difficult and disjointed notions of access and sharing. In such a context, LangSec-based vulnerabilities threaten catastrophic loss of privacy for all users of the system, likely even of non-vulnerable applications. This is a tragic state of affairs, but is thankfully not inevitable! We present the Sandstorm system, a capability-based, private-by-default, tightly-sandboxing, *proactively secure* environment for running web applications, complete with a single, pervasive sharing mechanism. Sandstorm, and capability systems, are likely of interest to the LangSec community: LangSec bugs are *mitigated* through the robust isolation imposed by the Sandstorm supervisor, and the mechanism of capability systems offers the potential to turn difficult authorization decisions into LangSec's bread and butter, namely syntactic constraints on requests: *every well-formed request which can be stated is authorized*. We present aspects of the Sandstorm system and show how those aspects have, by building systematic protection into several levels of the system, dramatically reduced the severity of LangSec bugs in hosted applications. To study the range of impact, we will characterize addressed vulnerabilities using MITRE's Common Weakness Enumeration (CWE) scheme.**

I. Introduction

Sandstorm[1] bills itself as "an open source operating system for personal and private clouds." Key among its features is proactive, robust inter-application and inter-user default isolation: users can install a wide variety of applications, of various degrees of trust-worthiness, and should be confident that any malicious or errant application will be limited in the damage it can do. At the same time, Sandstorm offers an expressive framework for explicitly sharing access to resources and auditing how far resources have been shared. It is important to note that Sandstorm is *not* a "Web Application Firewall": the applications under its supervision run largely as is, and Sandstorm simply routes requests and responses without altering the display content thereof.

The Sandstorm organization maintains a collection of "Security non-events"[2] which enumerates many Common Vulnerabilities and Exposure (CVE) identifiers [8] that have allegedly been completely obviated or largely mitigated by Sandstorm's isolation infrastructure. The Sandstorm project undertook an effort to collect all 2014 and 2015 CVEs affecting a subset of the applications that had been ported to use Sandstorm, namely Etherpad, WordPress, Roundcube, and Tiny Tiny RSS; for WordPress specifically, CVEs were selected only from those with an associated "severity score" of 6 or more (out of 10, the most severe). This resulted in a list of 20 CVEs. Additionally, 27 of the 224 CVEs reported against the Linux kernel between 2014 and 2016 are enumerated; we presume that the remainder have simply not been evaluated. The present paper reviews these CVEs, evaluating the protection afforded by Sandstorm and affirms the Sandstorm project's claim that "95% of (application) security issues automatically mitigated, before they were discovered."[3] We will categorize the CVEs using MITRE's Common Weakness Enumeration (CWE) scheme [9] to study the range of Sandstorm's impact.

This paper primarily hopes to stimulate further study both of the applicability of language security to *authorization*, beyond its traditional application to input languages and, dually, of the applicability of system design for mitigation of traditional LangSec vulnerabilities. We argue that many vulnerabilities present in software are only as severe as they are due to language security issues pertaining to (implicit) authorization assertions. We argue that the object-capability model of authorization [1, 2] offers a more LangSec-friendly approach which is less prone to adversarial tampering and misinterpretation.

A. Authorization

Broadly, authorization is a time-varying relation between agents and entities in a system and might be summarized as "who can do what to whom when?" Lampson, in [6], formalizes this (without an explicit notion of time variance) using the concept of an *access matrix*, a hypothetical data structure indexed by both agent ("domain") and target entity ("object") and storing the "rights" (or "permissions") afforded to that agent about the target entity. While one might imagine a universal ontology of rights, it is more common to have the type of rights be defined by the target entity. Perhaps the most popular rendering of this data structure in real systems is as "Access Control Lists" (ACL) whereby each object has associated with

[1] https://sandstorm.io/

[2] https://docs.sandstorm.io/en/latest/using/security-non-events/; this paper specifically references the revision as of March 16, 2016 made as Sandstorm commit https://github.com/sandstorm-io/sandstorm/commit/1a06f547.

[3] https://sandstorm.io/news/2016-02-29-security-track-record

it a list of agents and their rights to that object. Capability systems use a transposed view, wherein each agent is associated with its list of rights to objects.

An important and subtle concept when discussing authorization is *ambient authorization*: authority granted implicitly, used without explicit justification. On a POSIX system, for example, any process may grow its memory allocation via `sbrk`; it may request to `open` files relative to the *system-wide* root directory; etc.[4] By contrast, we could imagine a POSIX-like system where the only way to open a file was to name it relative to an existing file descriptor, or where `sbrk` was a message written to a (kernel-provided) file descriptor. In this modified POSIX, some processes may retain access to the real root directory and/or the ability to invoke `sbrk`, but it becomes possible to construct processes which do not, by virtue of not possessing the requisite *explicit* witness of authorization, that is, without the *capability* to act.

B. Object Capabilities

Object-capabilities, in the sense used here, are references to a target object together with an enumeration of rights to that object [1, 2]. In OO terms, they can be considered as references to an implementation instance of an *interface*, a *subset* of the methods exported by the pointed-to object. Crucially, capabilities are first-class components of the system and may be passed between agents, bestowing a subset of one agent's authorization upon another. The issuer of a capability may *revoke* it, making it inert, no longer authorizing anything. Capabilities may be *delegated*, copying the authorization of one capability to a new capability which may be revoked by the delegator. Revoking a capability revokes all its copies and any delegated versions, recursively. Rights may be *attenuated* (i.e., reduced) during delegation; e.g., having read/write access allows delegation of read-only access in addition to read/write access.[5]

Secure object capability systems must use *tamper-resistant* and *unforgeable* capabilities. The former property can be summarized as requiring that an agent holding a capability may not change anything about that capability: there should be no general mechanism for the agent to change which object is referenced or the rights afforded by this capability. The latter property requires that, at creation, only the creator of an object holds a reference to the created object, and the only way for another agent to gain a reference is for one to be *explicitly passed*.[6] These are very LangSec-esque properties. From a formal language perspective we can imagine an infinite collection of reference symbols, with creation operations returning never-before-seen elements and all other operations only able to pass around existing symbols. In practice, a variety of techniques, sketched below as needed, are used to observably emulate or approximate such an infinite set.

C. Capabilities and LangSec

Capability designs and LangSec approaches are both *proactive* approaches to security: they seek to render particular classes of vulnerabilities impossible by construction. LangSec seeks to make correct *input validation* a fundamental part of *input parsing*, rather than as a second step, so that parsers correctly "imbue input with trust" [4]. Capability systems are designed with the goal of enabling very fine-grained authorization and pushing this authorization into the computational substrate, rather than as a second step. We believe the requisite effort to properly eliminate ambient authority within a capability system, so that the fine-grainedness is not merely illusory, is in kinship with the LangSec call to eliminate "weird machines";[7] the goal of capability engineering is to simplify the input validation effort of computational agents, to make it the case that *any well-formed request which can be stated is authorized*. Thus, after checking that a request is syntactically well-formed, there should be no need for additional authorization logic. We contend that systems designed with this goal in mind will have better security by construction: certain failure modes can be entirely ruled out and large classes of vulnerabilities, LangSec-esque and otherwise, will be reduced in severity.

D. Typical Multi-user, Multi-application Web Hosting

In order to properly contrast Sandstorm with what has come before, we must spend some words detailing the dire typical state of multi-user, multi-application web application hosting. In such a setting, applications live side-by-side, in the same file hierarchy as each other, and are invoked by a single shared web server; persistent data for multiple users and multiple applications cohabitate in the same database engine. Inter-application isolation is achieved by *coarse-grained* mechanisms, available only to system administrators, including running applications as different (kernel) users, setting permissions in the filesystem appropriately, providing multiple (database) users within each shared database, etc.

Because isolation mechanisms require privilege far in excess of what we intend for hosted applications, applications must typically do without and each manage several users' data. Intra-application isolation is implemented separately, within each application (e.g., files uploaded by one user are readable by the application when acting on behalf of another user; it is up to the application to ensure that it will not read that

[4] This is a remarkably simplified view. POSIX defines `rlimits` for limiting acquisition of memory and open files, SUSv2 defines `chroot`, a privileged mechanism of creating process hierarchies whose root directory is a subdirectory of the system-wide root directory, etc. We trust that readers will forgive us the imprecision for the sake of example; the point remains that it is always possible to request a `sbrk` operation and there is always *some* root directory for a process.

[5] The target and rights of a capability are sometimes viewed as constants, set once at capability creation time and unchanging until the capability is revoked or discarded, but some systems permit agents to change these properties. Smalltalk [3], an early OO language, permits objects to "become" other objects, transparently forwarding all references; Smalltalk permits this generally, allowing any agent to replace any object to which it has a reference, but a secure version could be engineered. Coyotos [13] has no built-in notion of rights and instead attaches to each capability a "protected payload" whose semantics are up to the recipient object.

[6] One may, perhaps, allow components of the *trusted computing base*, such as a trusted garbage collector or, in this case, the Sandstorm core, to have references to created objects, as well. These components are assumed to not expose their references to untrusted agents of the system. In practice, this is, historically, somewhat difficult but not impossible.

[7] http://www.cs.dartmouth.edu/~sergey/wm/ and http://langsec.org/occupy/

file if it is not intended). Absent privilege, there are no or few system-level mitigations available: one typically cannot dynamically create a new (kernel or database) user for each application user, file system permissions are granular only at the level of applications, etc. Worryingly, if an application is vulnerable to remote code injection attacks, the application often has nearly arbitrary access to the network, allowing any point of compromise to become a springboard for further compromises. Further, the (shared) web server and kernels hosting the application process are themselves large, complex pieces of software with large surface areas which may be probed for vulnerabilities by injected code.

With that bleak world in mind, let us proceed on to the Sandstorm system and particular classes of vulnerabilities mitigated by its architecture. In the next section, we will focus on server-side vulnerabilities, largely neglecting the client and instead focusing on the effects of extremely strong sandboxing, which make up the bulk of Sandstorm's "security non-events." The section thereafter will turn to concerns of client-server, inter-application, and multi-user interaction.

II. Granularity in Sandstorm

We begin our discussion by focusing on an individual instance of an application hosted within Sandstorm, focusing on Sandstorm's unusually tight sandboxing of application code. The capability system used for communication with and within Sandstorm obviates the need for much of the traditional (POSIX) machinery (e.g., the BSD sockets API) and thereby enables what might otherwise be seen as impossibly prohibitive sandboxing, which is, as we now show, able to mitigate many LangSec-style bugs.

A. Sandstorm User Interface

The Sandstorm user interface is focused on a list of *grains*: instances of installed applications. While users may install new applications, create new grains of installed applications, and delete grains they have created, the most typical action is to search for and select an existing grain (to which the user has access) for further interaction. Like a typical web-server, Sandstorm launches the application and connects it with the user's web browser.

By way of concrete example, to begin authoring a document such as this in a Sandstorm hosted installation of ShareLaTeX, a collaborative LaTeX document editor,[8] this author authenticated to a local deployment of Sandstorm, instructed the server to retrieve the ShareLaTeX application from the Sandstorm market[9] and install it, and then requested a new ShareLaTeX grain. That grain hosts only this document and does not concern itself with others. When returning to the task of writing this document, it sufficed to select the appropriate grain from the list and wait for ShareLaTeX to start up. While that grain existed, the deployment of Sandstorm, of course, concurrently ran other ShareLaTeX grains as well as grains of other applications entirely, to which this grain had no access.

We will discuss sharing access in Sandstorm more later, but a few points are important here. When a user creates a grain, Sandstorm ensures that she alone has a capability to access it; this capability conveys full authority to the grain. That is, *all grains start off private by default*. Any user with access to given grain may ask Sandstorm to generate a URL that acts as a capability to access the grain. These capabilities are, of course, revocable and their rights are a *dynamically* attenuable subset of the creator's.[10] Sandstorm has largely-unused plumbing features to enable delegating access on a more precise model than per-grain sharing; since these features, collectively known as the "powerbox", are not used yet by any apps on the Sandstorm app market, this paper does not consider them.

B. The View Within A Grain

Sandstorm provides proactive security by offering different semantics than we sketched in § I-D; in a sense it is the notion of (POSIX) software sandboxing taken to an extreme.[11] Relatively new features of the Linux kernel, such as `seccomp-bpf`[12] private mount name-spaces[13] and network namespaces[14] are used to reduce the application's access to the system and the network: only a subset of system calls are permitted, few "device" files exist (and no more may be created), the only visible network interface(s) are loopbacks, the application software is mounted read-only, and only per-instance resources are mounted read-write. The sole mechanism for communication with the external world is a Cap'n Proto[15] socket to the Sandstorm supervisor, running outside the sandbox.[16] Thus, the ambient authority of an application hosted within Sandstorm is dramatically reduced by comparison to a typical UNIX-style process. One could imagine this entire exercise as a dramatic *reduction* in the power of the language that the application speaks with the system: fewer operations are possible on fewer objects than

[8] https://www.sharelatex.com/

[9] Direct uploads of packaged applications are also possible; the marketplace is a *convenience*, not requirement.

[10] Two subtle points merit brief mention. First: sharing URLs created this way are *not* directly usable by application grains, so, for example, even if one is using Roundcube under Sandstorm to send one's mail, the Roundcube grain does not come to hold capabilities transiting it. Second: Sandstorm's capability system will transitively re-attenuate rights when it needs to. That is, if A grants read/write access to B and B grants read/write access to C, but then A dynamically attenuates the capability held by B to be read-only, then the capability held by C will also be read-only.

[11] See https://docs.sandstorm.io/en/latest/using/security-practices/.

[12] See Linux's `Documentation/prctl/seccomp_filter.txt`

[13] Linux's `Documentation/filesystems/sharedsubtree.txt` may be the appropriate starting point; this feature is remarkably under-documented. For an overview of name-spaces in UNIX-like systems more generally, the curious should start with [11].

[14] See the manual page for Linux's `clone(2)` system call, and in particular the `CLONE_NEWNET` flag.

[15] https://capnproto.org/

[16] The Cap'n Proto serialization and de-serialization code is automatically generated from schema descriptions and can be robustly tested independently of its role in the target application, following good LangSec practices. Live capabilities in Cap'n Proto are rendered on the wire as integers, with no other structure or meaning to any agent other than the other participant in the connection; each end of the socket maintains tables mapping wire representations to pointers. A consequence of this representation is that live capabilities cannot be transferred without the active participation of other end.

in the traditional hosting configuration, thereby resulting in a dramatically simpler kernel-side protocol state machine [12].

C. Granularity As Editorial Decision

In light of the above view, one could intuit that exploits of application vulnerabilities are already limited: they are confined to a grain and to the resources available to that grain. Under fine-grained instances, many vulnerabilities turn out to not be significant in practice. At one extreme, grains may host *individual documents*; at the other extreme, an entire file storage system (e.g., Davros[17]) may be a single grain. The decision of what constitutes a grain is thus up to the application author and/or packager. Currently, Sandstorm functions under a per-grain access model, so we see grain boundaries drawn around sets of objects which are closely related and which have approximately covarying permissions. Especially simple cases include when users have read-only or read/write access to the entire ensemble, but options such as "read-only for document objects and read/write comments" often found in document editors are also possible. Thus, for simple applications like ShareLaTeX, there is a natural, maximally-fine-grained packaging which puts each document (and its multiple source files) in an individual grain. Similarly, wiki engines tend to have rights that apply to all pages within, perhaps also with limited "administrative" access for some users, making it appropriate to use a grain for related pages within some domain of administration, but inappropriate for multiple administrative domains to share such a grain.

When sharing of individual objects, as defined by particular applications, becomes available in Sandstorm (as the "powerbox"), we should expect grain boundaries to settle wherever the risk of accidental exposure or mishandling of material by the application within the grain outweighs the cost of crossing the grain boundary. Users may be advised, for example, to have separate file-hosting grains for widely-shared family photographs and for narrowly-shared financial documents, so that Sandstorm's inter-grain isolation continues to provide security even if an application's intra-grain isolation fails.

D. Security Impact

1) Path Traversal Vulnerabilities: Several of the vulnerabilities mitigated by Sandstorm fall under "CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')." Path traversal vulnerabilities are authorization language security issues: the existence of such a vulnerability requires that an agent be able to access a path outside the set intended by the application's authors for this agent in spite of whatever protections are in place within the application. The fine-grained isolation approach afforded by capability systems defangs LangSec bugs; the goal is that encapsulated software cannot carry out actions that significantly impact user data privacy or security, even if it incorrectly parses input file paths.

The mechanism of action of a (POSIX system) path traversal vulnerability provides an example of ambient authority: as hinted at in § I-A the POSIX API does not offer a mechanism for an unprivileged application to confine its actions in the filesystem to a sub-tree. *Absolute paths* refer to an (explicitly named) object relative to an *implicitly named, omnipresent* root directory; even the (relatively) modern POSIX `openat` system call is specified to interpret absolute paths relative to this root rather than the provided `dirfd`. Moreover, by definition, every directory has a reference to its parent directory available, as "`..`".[18] Therefore, if we mean to manipulate or expose only a particular directory or subtree of the filesystem in a context where arbitrary paths may be given, we must be careful to normalize away and eliminate these possibilities that would escape the subtree, as the kernel will not enforce our desire for paths to be confined.

All of that to say, path traversal vulnerabilities are *also* in the domain of (traditional) LangSec: canonicalizing away "`..`" and preventing the initial "/" of an absolute path are obviously syntactic matters. The design of the POSIX API, with its lack of confining operations, all but ensures that parsing bugs are security vulnerabilities.[19] We now discuss the path traversal vulnerabilities in Sandstorm's non-events list.

a) CVE-2015-0933: ShareLaTeX did not properly restrict LaTeX's `\include` facilities, allowing users to read arbitrary files. The result, again, is arbitrary client-controlled reads of files on the server. The CWE ontology appears to lack a designator for this kind of directive-based file-read vulnerability in general; perhaps "CWE-22" ("Path Traversal") remains the most appropriate designation, despite its lack of specificity. In order to exploit this vulnerability, a user requires write authority to a document, to insert an `\include` directive. Recall from § II-C that each ShareLaTeX document runs in its own grain on Sandstorm. A user with write authority to that grain's single document also, by necessity, has read authority to that document and the ShareLaTeX application software itself, and by construction there are no other files in the grain. Thus we see a common end of vulnerabilities when run under Sandstorm: users can attack themselves but have no ability to forge capabilities to other grains to attack.

b) CVE-2015-3297 and -4085: Etherpad, a real-time collaborative document editor,[20] normalized then transformed client-controlled strings representing filesystem paths. In CWE terms, the first bug is an instance of "CWE-172: Encoding Error", specifically "CWE-180: Incorrect Behavior Order: Validate Before Canonicalize." In the second case, "CWE-182: Collapse of Data into Unsafe Value" appears to be the correct designation. In both cases, clients can craft paths to reference,

[17] https://github.com/mnutt/davros/

[18] This reference may not be removed and the only way to prevent its traversal appears to be to prevent search of the directory or its parent as a whole, which is rather extreme.

[19] For brief contrast, a more capability-system style design would be that *all* path traversals are confined relative to an explicitly stated directory and that applications would explicitly hold "cursors" into the system; there would be no "`..`" references in the filesystem itself. While such a model may make it difficult to *ask* if a particular file is below some path in the file system, there is very little call to know the answer. Oddly, many UNIX shells manually interpret "`..`" internally, to prevent user confusion in the face of links [10].

[20] http://etherpad.org/

and cause the server to read for the client, arbitrary files on the server system.[21] In Sandstorm, each Etherpad document is a separate grain, so as with ShareLaTeX, users could exploit pads to which they have access but no others.

c) CVE-2015-5382 (AKA CVE-2015-8794): The Roundcube web-mail program[22] contained an inadvertent implementation of `cat`: an authenticated user could request an arbitrary absolute file name and the server would read back the contents.[23] This may be an example of "CWE-36: Absolute Path Traversal" (if the intent was to allow only relative paths) or "CWE-73: External Control of File Name or Path" (if restrictions were intended but never implemented). Again, clients can construct requests to read arbitrary files on the server, including other users' mail. Each Sandstorm grain of Roundcube manages but one user's mail, rendering this moot: a user can read only that which they could already read.

We can see that the path traversal vulnerabilities take many forms, but that the *unusually fine-grained* sandboxing of Sandstorm *mitigates* the problem. Applications which, unlike the examples given above, continue to maintain differing permissions for different subsets of agents and objects will continue to be at risk of bugs which unintentionally bestow rights to agents, including accidental information disclosure and unintended path traversals in particular. While one can hope to mitigate the damage, ultimately, sandboxing cannot eliminate the burden of correctness within a grain. However, often (admittedly, not always), the need for such difficult authorization juggling is precisely the lack of fine-grained partitioning! All three applications listed above, when run outside of Sandstorm, attempt to restrict users to access only their objects (i.e., documents and email); wrapping these applications in a capability system removes this need as, within a grain, there is no longer any information which a user is not authorized to see.

2) Other Information Disclosures: CVE-2015-5383 is a case of "CWE-532: Information Exposure Through Log Files" (as well as possibly "CWE-117: Improper Output Neutralization for Logs") in Roundcube; log files were visible to unauthenticated users. The Roundcube log occasionally contains authentication cookie text; this would allow anyone to impersonate a legitimate user.[24] The Sandstorm port of Roundcube hosts only one user's mailbox, largely mooting this vulnerability. Generally speaking, log leakage from within a Sandstorm sandbox is unlikely to be catastrophic due to the fine-grained nature of instances and Sandstorm's access control.[25] However, Sandstorm sharing URLs intended for user distribution may be unintentionally shared through such leaks (e.g., if Roundcube improperly included one email body in a message to another address); the only silver lining there are the possibility of auditing to reveal the flow of (unintentional) delegation and the ease of revocation.

3) Code Injection: In addition to mitigating "passive" attacks like path traversals, fine-grained sandboxing and the attendant permissions management performed by the Sandstorm core largely moots many code injection exploits wherein an attacker comes to run adversarially-chosen instructions server-side. Often, the only possible attackers are users with full access to the grain, and the Sandbox should prevent cross-grain attacks.

CVE-2015-0934 is a case of "CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')," wherein ShareLaTeX could execute arbitrary commands when given file names involving back-quotes. Only users with write access can trigger this bug, and each grain contains only one document, rendering it useless.

TinyTinyRSS (TTRSS), a RSS reader,[26] suffered a SQL injection ("CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')") attack,[27] enabling authenticated users to take complete control of the server. On Sandstorm, each TTRSS grain is typically *unshared*, the sole capability to it held by its creator.

Last, CVE-2014-5203 in the WordPress web framework[28] is a classic language security problem: it is a case of "CWE-502: Deserialization of Untrusted Data" due to "CWE-354: Improper Validation of Integrity Check Value;" in particular, the integrity check was done after deserialization.[29] (Further, the hash used was MD5, making this "CWE-327: Use of a Broken or Risky Cryptographic Algorithm," too.) This bug, too, required write access to exploit.

As with path traversal vulnerabilities (§ II-D1), these particular vulnerabilities are not terribly interesting: they either require write access or are to grains typically kept private to a user. In principle, however, code injection attacks *could* be used to escalate permissions. If, for example, read access to one object is sufficient authority to invoke the code injection bug, then a user intended to have only that authority in fact has read/write access to the entire application.

4) Kernel Vulnerabilities: Sandstorm is a shared-hosting infrastructure: it potentially hosts applications on behalf of multiple users. As such, in addition to mitigating attacks against applications by users or third parties, must protect applications from other applications resident on the same host. Because all communication is forced through the Sandstorm core, applications see each other as foreign agents, so there is no risk

[21] The fix to the first CVE is https://github.com/ether/etherpad-lite/commit/9d4e5f6 and corrects backslash characters' replacement with slashes (an attempt to reconcile different platforms' directory separators) after checking that the path had not escaped the server's intended directory. (In passing, we note that CVE-2015-3309, not enumerated on the Sandstorm non-events page, is a follow-up to this being an incomplete fix, missing identical code elsewhere in the software.) The second is fixed by https://github.com/ether/etherpad-lite/commit/5409eb3; here, a client-controlled suffix was concatenated to a server prefix with only the *first* occurrence of "`..`" eliminated and no validation was performed.

[22] https://roundcube.net/

[23] http://trac.roundcube.net/ticket/1490379

[24] http://trac.roundcube.net/ticket/1490378

[25] Once the aforementioned "powerbox" features are more widely used, grains will hold capabilities to other objects in the system. These capabilities will be bound by Sandstorm to be useful only to the grain that holds them, making them un-leakable by information disclosure.

[26] https://tt-rss.org

[27] No CVE is assigned for this vulnerability; see http://security.szurek.pl/tiny-tiny-rss-blind-sql-injection.html;

[28] https://wordpress.com/

[29] http://openwall.com/lists/oss-security/2014/08/13/3

of inadvertent cross-application escalation of privileges due to seeing requests from loopback addresses or similar (a common form of "CWE-266: Incorrect Privilege Assignment"). However, the local kernel is a high-value target for malicious applications: successful compromise offers unlimited control of the entire hosting machine, including the Sandstorm supervisor and all applications running there. The Sandstorm security non-events page enumerates 24 CVEs completely obviated by Sandstorm's use of system-call filtering (`seccomp-bpf`). This filter is obviously effectively reducing the Linux kernel's attack surface.

However, system call filtering is not a panacea: at least three kernel-vulnerability CVEs remain possibly applicable (CVE-2014-9090 and -9322, possibly best categorized as "CWE-755: Improper Handling of Exceptional Conditions," and CVE-2016-2069, a particularly exotic form of "CWE-416: Use After Free"). These bugs are fundamental problems in the kernel's state machine and no user-land mitigation is possible. Some small solace can be taken from the apparent difficulty of reliable exploitation.

III. Multi-User Sandstorm

A. Sandstorm Multi-User Interface

Revisiting the ShareLaTeX example above, when the owner of a grain (hosting a single document, recall) is ready to share the document with collaborators (in a read-write manner) or for review (in a read-only manner), she asks the Sandstorm core itself, not ShareLaTeX, to grant access to this grain and is given a URL representing the appropriate level of access.

A single-document ShareLaTeX grain is particularly amenable to the kinds of permissions that Sandstorm understands well: there is just one permission bit (write or not) beyond access to a grain. Thus, the Sandstorm port of ShareLaTeX no longer maintains its own notion of user authentication and user authorization, though it retains the notions of concurrent access to one document by multiple agents and the authority required by particular requests (i.e., read or write). Our goal that all well-formed requests are authorized means that the sole check of a request to determine authority is one of *syntax*: any request to mutate the document must be stamped with a Sandstorm header that indicates that the requestor has write access. If this is so, it must be authorized according to the Sandstorm core capability system; if not, the request is *not well-formed* and can be aborted as such.

Sandstorm's capability system contains an unusual feature intended to make human understanding of delegation simpler: if a user comes to hold multiple capabilities to a grain (or object, more generally), their rights are the *union* of the rights conferred by any of those capabilities. Any request made by that user to that grain will be labeled with this union. Revocation or dynamic attenuation will prompt the recomputation of a user's rights, as expected. The net effect is a familiar user interface: when a user visits a grain, they are always able to use all permissions given to them, even if those permissions were granted separately. This glosses over the capability implementation details of Sandstorm.[30]

B. Implementation

When references must be serialized for presentation, such as when users wish to share a grain by URL, Sandstorm provides to the users an opaque, unforgeable sharing URL. These long, random bit-strings are the computational approximation of the infinite formal language of § I-B and their opacity ensures an absence of LangSec bugs such as path traversals.[31] The Sandstorm supervisor is responsible for mapping these URLs back to capabilities within the system (and is the sole agent in the system that can), including checking for validity and revocation. This life-cycle management is entirely implemented in the Sandstorm core;[32] applications are largely oblivious and are simply told what (application-defined) rights are associated with every incoming request.[33]

Apps that run on Sandstorm can leverage these trusted headers to make it easier to audit them for security. Note that there is a syntactic LangSec opportunity here! The application (and its API) should be structured so that it is easy to deduce *from the request* what rights are required, so that this can be compared against the set of rights asserted by Sandstorm. In a traditional web application, this is often the case, by recommendation: `POST` and `PUT` verbs are used for things requiring some kind of mutation authority while `GET` is typically used for read-only actions, and, moreover the path of a URL often straightforwardly corresponds to more refined rights (e.g. objects within an `/admin/` path may require greater permission to access than other paths). Sandstorm replaces the application's "weird machine" which maps requests to users to authority with a (hopefully) simpler machine which maps requests to required authority which is then *checked against a*

[30] This does increase the risk of a "confused deputy" attack [5], in which a user is tricked to into carrying out operations on a grain by another agent that does not have the requisite authority. The tradeoff in favor of simplicity is probably justified in that we may expect humans to exercise better judgement than autonomous software agents. This unioning of rights does *not* happen for capabilities held by software.

[31] There are emerging interfaces in Sandstorm for sharing different views of an application, including access to specific objects (recall § II-C); here, Sandstorm hides the application's potentially insecure object identifier (e.g., row ID in a database) behind its unforgeable URL.

[32] A major design requirement is that Sandstorm be aware of all capability passing within the system it constructs. The restrictions on tokens ensure, among other things, that applications which are permitted to exchange data (using capabilities) cannot exchange capabilities without explicit authorization. This mitigates accidental leakage and enables auditing of permissions, and is *crucial* for enforcing the "*-property" of confinement [7]. That is, merely knowing the *bytes* of a Sandstorm capability representation is *not necessarily sufficient* to use that capability.

[33] Sandstorm additionally informs the application of the "display name" of the user making the request and, for legacy applications' use, provides a hash of the user's identity as a surrogate "user name". While not viewed as ideal by all Sandstorm developers, some applications' Sandstorm ports use this information to update the application's authorization database and then run existing code unmodified, as if the user had logged in.

trusted part of the request itself.[34]

1) Authorization Vulnerabilities: CVE-2015-2298 is an information disclosure vulnerability in Etherpad whereby a request to export a pad's contents by its server-side identifier exports the contents of all pads whose identifiers contain the requested one as a substring, regardless of the requesting agent's authority.[35] It is, in one sense, a standard LangSec-esque bug, a failure to normalize input into a lookup so that it carried out exact, rather than substring match, but we prefer to think of it as a *missing autorization check*. Like the path traversals in Etherpad, this bug is completely neutralized in Sandstorm: each grain has only one document and so even overzealous selection code can only select that one document. A user with sufficient privileges to request export already has sufficient rights to read that document directly.

CVE-2015-9038 is a WordPress vulnerability in which an *unauthenticated* attacker can induce WordPress to make HTTP requests of itself. WordPress, like many other programs, recognizes these "loopback" connections and endows them with additional rights under the premise that only a machine administrator could cause them to be made. That is, these connections are given ambient authority merely by virtue of their source, and requests from the application itself, naturally, share that source. This particular vulnerability is exploitable under Sandstorm only by users holding editor or administrative access to the WordPress grain, largely rendering it unimportant: an attacker gains no more authority than they previously had. Applications should be discouraged, as a general rule, from providing ambient authority to loopback connections and should instead ensure that there is some explicit token indicating that the request came from an administrator.

While the above bugs have straightforward resolution under Sandstorm, missing or broken authorization checks are a real risk. Sandstorm's simplified story of being able to replace all authorization checks with simpler syntactic-well-formedness tests on requests has the potential to make it much easier to not only audit, manually or mechanistically, that the checks are in place, but, potentially, to automatically generate them from high-level specifications. For example, it may be possible to verify that all code paths leading to SQL `INSERT` statements first check the trusted Sandstorm headers.

2) Authentication Vulnerabilities: Three of the CVEs documented (CVE-2015-9037, CVE-2014-9033, and CVE-2014-0166) are in code involved in authenticating users or associated functionality within WordPress. CVE-2014-9033 is a CSRF vulnerability, a class we will address in the next section. CVE-2014-0166 and CVE-2015-9037 both arise as a result of PHP's implicit coercion for comparisons ("type juggling");[36] here, an MD5 output from a password hashing function could be coerced and compared incorrectly ("CWE-187: Partial Comparison"). However, the vulnerable code is simply *unused* on Sandstorm, where user authentication and authorization logic is centralized to the core and therefore reusable across all applications and *independently testable*.

IV. Sessions as Capabilities

The Web is a scary platform on which to build secure software, as it was largely not designed with security in mind; the attack surface of a modern web application is not only the endpoints running on the server but also its JavaScript running client-side (as well as any JavaScript that could be loaded from the same host, even from a different application, due to the web's "same origin policy"[37]), the client JavaScript engine, any unintended interactions between resources within the client, etc. We discuss some of the effort Sandstorm invests in closing Web-specific bugs, the CVEs mitigated and not, and briefly look towards future plans for increasing platform security.

A. Implementation

In order to ensure that only the anticipated user may access a particular grain, every *session* (i.e., an individual user's access to individual grain) is given a cryptographically-random hostname and old names are expired from the Sandstorm core quickly; Sandstorm itself uses a session cookie to authorize access to a particular generated hostname.[38] Thus, despite Sandstorm not altering the display content of grains, a measure of unforgability is introduced into all requests. While the hostnames are leaked by the client in clear-text (due to DNS and TLS SNI), they are still difficult for *off-network-path* attackers to forge. The session cookie is guarded against evesdroppers by wire cryptography and against exfiltration by the user's web browser. While not a complete solution, these tricks do mitigate real security vulnerabilities.

B. Security Impact

1) CSRF vulnerabilities: A popular attack vector against (web) applications is "CWE-352: Cross-Site Request Forgery," (CSRF or XSRF) wherein a resource provided by one server (A) causes the client to make requests of another server (B). This kind of linkage is fundamental to the Web's success, but care must be taken to ensure that *privileged* requests (of B) are not merely *authorized* (i.e., made by a client with sufficient access to B) but *deliberate*. Failure to properly vet requests is rampant; the Sandstorm security non-events page enumerates five in its sampling (CVE-2014-5204, CVE-2014-5205, CVE-2014-9033, CVE-2014-9587, and CVE-2015-5731).

CSRF vulnerabilities can be viewed in at least two lights. First, as LangSec bugs: (web) applications, with their open

[34] When all is said and done, a HTTP application running under Sandstorm's supervision is given a "X-Sandstorm-Permissions" header as part of its requests. This header is generated by the Sandstorm HTTP bridge from information carried within the Cap'n Proto encoding of the request from the supervisor and, ultimately, its content derives from the capabilities held by the agent making the request. Crucially, the client may not influence the value of this header; it would not transit the HTTP application component of the Sandstorm core. Its format is a comma-separated list of alphanumeric identifiers; it is designed to be easy to unambiguously parse.

[35] https://github.com/ether/etherpad-lite/commit/a0fb6520

[36] As documented at http://php.net/manual/en/language.types.type-juggling.php. The author contends that this "feature" could not have been better designed to produce LangSec issues.

[37] https://www.w3.org/Security/wiki/Same_Origin_Policy

[38] See "Client Sandboxing" at https://docs.sandstorm.io/en/latest/using/security-practices/.

channels of communication (where nearly arbitrary clients may make requests) must ensure that multi-stage communication with their clients recognizably bind together the client's next answer to the application's previous response. A web application must generate content for the client that will, in turn, cause the client's next request to attest continuity of conversation. That is, CSRF protection amounts to a recognizer of requests that are authorized to convey a user's authority. Second, we may also understand CSRF attacks as a failure of the web to offer unforgeable capabilities to make requests bearing the user's authorization: the attacker is able, with cooperation of the client (and its authorization by B) to convert a sequence of bytes into a capability they should not hold. Indeed, popular mitigations are excitingly similar to those used by capability systems; most often, CSRF protection involves embedding cryptographically random nonces in ways that will be echoed back to the server by legitimate requests but will be omitted or incorrect in forged requests.

Sandstorm's creation of a unique hostname for each session implies that a would-be attacking server A must either guess the hostname (impractical if Sandstorm's randomness is not severely flawed) or obtain the hostname by sniffing client traffic (requiring that the attacker be in a position to do so; CSRFs are powerful in most settings precisely because they do not typically have this requirement). Because CSRF requests are, ultimately, made by the user's client, the session cookie will be passed along with them. While less ideal than correct CSRF protection, the use of unguessable hostnames still raises the bar of exploitation and acts as a form of failsafe.

2) XSS attacks: The web as a platform is also vulnerable to so-called "Cross-site Scripting" (almost always abbreviated as XSS) attacks, designated "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')." The simplest sub-class of XSS are "reflected XSS" attacks in which a malicious agent crafts a URL to a vulnerable server in such a way that the server embeds attacker-controlled content into the material the server feeds to the client, escaping the intended grammar of the server's response: a classic LangSec problem. When the client loads this URL, it will consider the provided material to be server-originated, and so any scripts therein will run with appropriate privileges. The Sandstorm security non-events page contains two reflected XSS attacks (CVE-2015-2213 and CVE-2015-5381 (confusingly, also known as CVE-2015-8793)), mitigated by Sandstorm in the same way as CSRF requests: the malicious agent must know the name of the vulnerable application host, which is unlikely.

The third XSS attack on the security non-events page is CVE-2015-1433 and is correctly listed as *not* mitigated at all. This is a vulnerability in Roundcube's display of email to clients, allowing attackers to email victims JavaScript that will be executed when the mail is rendered.[39] As Sandstorm does not alter the content served by applications to clients, this exploit continues to work. Future work on Sandstorm will strengthen the *client-side* sandboxing, rendering similar XSS attacks unable to communicate outside the vulnerable application. Despite that, these more advanced ("non-reflected") XSS vulnerabilities represent a serious threat to all web applications, even when within Sandstorm's sandbox, as there appears to be little possibility of automatic, external mitigation that would eliminate unintended actions *within* the application given the design of the web platform.[40]

However, the platform is changing with time. The W3C has defined a "Content Security Policy" (CSP) mechanism[41] to address XSS vulnerabilities. CSP achieves its protection by allowing web resources to specify, in their response headers, origins for resources, including JavaScript and CSS, that may be legitimately referenced by the body content; of particular note is the ability to *disable* so-called "inline" and "`data` URI" resources. This raises the bar of XSS attacks: the response body often *deliberately* contains text from numerous sources (including email messages, as we see with CVE-2015-1433), requiring extensive LangSec mitigation to ensure that the composite XML/HTML document ascribes to the structure intended by the application, while the response *headers* are likely specified fully by the application itself.[42] Sandstorm does not yet support CSP, but support is planned.

V. Conclusion

Security and privacy of computation has, historically, largely been an afterthought, both within individual products, where having something that works is more important than something secure, and within the discipline of computing as a whole, where most efforts originated in small groups of mostly-mutually-trusting individuals. Most systems are, thus, either through design or oversight, "public by default." Capability systems, with their goal of minimizing ambient authority, offer a vision of a "private by default" world. Moreover, the mechanism of capability systems offers the potential to turn authorization decisions into easily verified syntactic constraints on requests: *every well-formed request which can be stated is authorized*. Sandstorm builds on such a system and allows users to track how far access has been shared (i.e., what incoming capabilities exist to each grain) as well as revoke access at any time. The strong reduction in ambient authority imposed on applications hosted within Sandstorm can further mitigate or eliminate many flaws in application software. Expressed another way, Sandstorm-like supervisors *potentially reduce the difficulty of developing secure multi-user software*.

MITRE's CWE vulnerability ontology is an interesting intellectual exercise but encourages a reactive, point-wise

[39] http://trac.roundcube.net/ticket/1490227

[40] It is similarly possible for an application to suffer from "non-reflected CSRF" vulnerabilities, too, if, for example, it is possible to embed attacker-controlled *relative* URLs into displayed documents.

[41] https://www.w3.org/TR/CSP2/

[42] Of course, the design of the web platform is such that response bodies make frequent *indirect* reference to other resources, i.e. by URL. `http` and `https` schemes do not have a mechanism for binding to the expected response body; for security-sensitive resources, such as JavaScript and CSS, this loss of binding between intended resource and name could be leveraged into code injection by an attacker. An emerging technology called "Subresource Integrity" (SRI; https://www.w3.org/TR/SRI/) allows pairing a URL with the hash of the expected response body.

approach to security, fixing individual exploits as they are discovered. Many of its entries are predicated on the existence of structured, malleable remote references, such as exposed local file paths. By reinforcing the notion that the problem is merely in how these references are filtered, we worry that CWE encourages perpetuation of dangerous designs rather than proposing alternatives which cannot, by construction, suffer from these flaws.

VI. Acknowledgements

We are deeply indebted to Asheesh Laroia, Kenton Varda, and Drew Fisher for their work on Sandstorm and many, useful suggestions for this paper. We would like to thank, as well, our anonymous reviewers for their excellent feedback.

References

[1] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966. ISSN 0001-0782. URL http://doi.acm.org/10.1145/365230.365252.

[2] M. S. Doerrie. *Confidence in Confinement: An Axiom-free, Mechanized Verification of Confinement in Capability-based Systems*. PhD thesis, Johns Hopkins University, July 2015. URL http://www.doerrie.us/assets/doerrie-dissertation-jhu.pdf.

[3] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[4] Robert David Graham and Peter C. Johnson. Finite state machine parsing for internet protocols: Faster than you think. In *Proceedings of the 2014 IEEE Security and Privacy Workshops*, SPW '14, pages 185–190, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5103-1. URL http://www.cs.dartmouth.edu/~pete/pubs/LangSec-2014-fsm-parsers.pdf.

[5] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22 (4):36–38, October 1988. ISSN 0163-5980.

[6] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974. ISSN 0163-5980. URL http://doi.acm.org/10.1145/775265.775268.

[7] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, 2003. URL http://zesty.ca/capmyths/usenix.pdf.

[8] *Common Vulnerability Enumeration: The Standard for Information Security Vulnerability Names*. MITRE Inc., 1999. URL https://cve.mitre.org/index.html.

[9] *Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types*. MITRE Inc., 2006. URL https://cwe.mitre.org/index.html.

[10] Rob Pike. Lexical file names in plan 9 or getting dot-dot right. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00. USENIX Association, 2000. URL http://plan9.bell-labs.com/sys/doc/lexnames.html.

[11] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, EW 5, New York, NY, USA, 1992. ACM. URL http://plan9.bell-labs.com/sys/doc/names.html.

[12] E. Poll, J. D. Ruiter, and A. Schubert. Protocol state machines and session languages: Specification, implementation, and security flaws. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 125–133, May 2015. URL http://cs.ru.nl/E.Poll/papers/langsec_draft.pdf.

[13] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel, 2004. URL http://www.coyotos.org/docs/osverify-2004/osverify-2004.html.