Twizzler-Security
A Capability-Based Security System for Twizzler

BSc Thesis

written by

Surendra Jammishetti

under the supervision of **Owen B. Arden**, and submitted to the

Examinations Board in partial fulfilment of the requirements for the degree of

Computer Engineering B.S.

at the *University of California, Santa Cruz.*

Members of the Thesis Committee:

Dr. Peter Alvaro
Dr. Andi Quinn

overall great start, I'd expend the intro a little bit, it's a little sparse and could use a few more things. future work + conclusion too.

run a spell check :)

more feedback throughout, feel free to take or ignore.

Abstract

Traditional operating systems permit data access through the kernel, applying security policy as a part of that pipeline. The Twizzler operating system flips that relationship on its head, focusing on an approach where data access is a first-class citizen, getting rid of the kernel as a middleman. With this data-centric approach, it requires us to rethink how security policy interacts with users and the kernel. In this thesis, I present the design and implementation of core security primitives in Twizzler. Then I evaluate the security model with a basic and advanced scenario, as well as microbenchmarks of core security operations. Lastly, I discuss future work built off this thesis, such as the incorporation of Decentralized Information Flow Control.

Contents

Chapter 1

Introduction

In mainstream operating systems, a omnicient and all-powerful kernel enforces security policy at runtime. It acts as the bodyguard, holding all i/o and data protected unless the requesting party has the authorization to access some resource. This tight coupling of security policy and access mechanisms works well since any access must be done through the kernel, so why not perform security checks alongside accesses? However, the enforcement of security policy starts getting complicated when we try to separate the access mechanisms from the kernel. This problem arises in a certain class of operating systems.

1.1 Data-Centric Operating Systems

Data-centric operating systems are defined by two principles [Bit+20a]:

1. They provide direct, kernel-free, access to data.
2. They have a notion of pointers that are tied to the data they represent.

Mainstream operating systems fail to classify as data-centric operating systems, as they rely on the kernel for all data access, and use virtualized pointers per process to represent underlying data. The benefit of this "class" of operating systems comes from the low overhead for data manipulation, due to the lack of kernel involvement. However, the mainstream security model fails to operate here as, by definition, the kernel cannot be in front of access to data. So, something new must be investigated.

1.2 Capability Based Security Systems

Capability-based security systems have a rich history in research, and offer an alternative approach to security, in opposition to the Access Control Lists of prevalent OS's [ZL09]. Boiled down, a capability is a token of authority, holding at minimum some permissions and a unique identifier to which "thing" those permissions apply to [Lev84]. This simple approach of having a "token", allows for a separation of the kernel's involvement in the creation and management of security policy. In a well-designed system, as we see in [Bit+20b] and described later, this allows users to completely create and manage security policy while the kernel is left to enforce it. This paradigm permits kernel-free access of data, while also guaranteeing security.

1.3 Our Contributions

In this thesis, I detail the fundamentals of security in the Twizzler operating system, and discuss how I implement and refine some of the high level ideas described in Twizzler [Bit+20a] and an early draft of a Twizzler security paper [Bit+20b]. Additionally, we evaluate these systems inside kernel and user space, using Alice/Bob scenarios and microbenchmarks. Code can be found in this Github tracking issue.

Chapter 2

Key Pairs

Key pairs in Twizzler are representation of the cryptographic signing schemes used to create a signed capability, as discussed in 3.1. We design the keypair objects to be agnostic towards the underlying scheme to allow for multiple schemes, as described in [Bit+20a]. This also helps with backwards compatibilty when adding new, more secure schemes, in the future. The keys are stored inside of objects, allowing for persistent or volatile storage depending on object specification, and allows for keys themselves to be treated as any other object and have security policy applied to them. This allows for powerful primitives and rich expressiveness for describing secruity policy, while also being intuitive enough to construct basic policy easily.

2.1 Abstraction

The `SigningKey` struct is a fixed length byte array with a length field and an enum specifying what algorithm that key should be interpreted as. Currently we use the Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] to sign capabilities and verify them, but the simplistic data representation allows for any arbitrary algorithm to be used as long as the key can be represented as bytes.

Additionally this specification allows for backward compatibility, allowing for an outdated signing scheme to be used in support of older programs / files. An existing drawback for backward compatibility is the maximum size of the buffer we store the key in. Currently we set the maximum size as 256 bytes, meaning if a future cryptographic signing scheme was to be created with a key size larger than 256 bytes, we would have to drop backwards compatibility. Sure this can be prevented now by setting the maximum size to something larger, but thats a tradeoff between possible cryptographic schemes vs the real on-disk cost of larger buffers.

2.2 Compartmentalization

To create an object in twizzler, you specify the id of a verifying key object so the kernel knows which key to use to verify any capabilities permitting access to the object. Since keys are represented as objects in twizzler, security policy applies on them as well, creating satisfying solutions in regards to key management.

Suppose for instance we have Alice on Twizzler, and all users on twizzler have a “user-root” keypair that allows for them to create an arbitrary number of objects. Also suppose that access to this user-root keypair is protected by some login program, where only alice can log in. This means that Alice can create new keypair objects from her user-root keypair. Since all **her** new keypairs originate from **her** original user-root keypair, only **she** can access the keys required to create new

signatures allowing permissions into **her** objects. It forms an elegant solution for key management without the involvement of the kernel.

hice -- we should talk more about this

Chapter 3

Capabilities

Capabilities are the atomic unit of security in Twizzler, acting as tokens of protections granted to a process, allowing it to access some object in the ways it describes. Colloquially a capability is defined as permissions and a unique object to which those permissions apply, but in Twizzler we add the signature component to allow the kernel to validate that the security policy was created by an authorized party.

Thus, a Capability is represented as follows:

```
struct Cap {
    target: ObjID,      // Object ID this capability grants access to.
    accessor: ObjID,    // Security context ID in which this capability resides.
    prots: Protections, // Specific access rights this capability grants.
    flags: CapFlags,    // Cryptographic configuration for capability validation.
    gates: Gates,       // Additional constraints on when this capability can be used.
    revocation: Revoc,  // Specifies when this capability is invalid, i.e. expiration.
    sig: Signature,     // The signature.
}
```

3.1 Signature

The signature is what determines the validity of the capability. The only possible signer of some capability is who ever has permissions to the signing key object, or the kernel itself. The signature is built up of a array with a maximum length and a enum representing what type of cryptographic scheme was used to create it; quite similar to the keys mentioned previously. The fields of the capability are serialized and hashed to form the message that gets signed, and then stored in the signature field. Currently we support Blake3 and Sha256 as hashing algorithms.

3.2 Gates

Gates act as a limited entry point into objects. If a capability has a non-trivial gate, which is made up of an offset field, and a length field, the kernel will read that and ensure that any memory accesses into that object are within the gate bounds. The original Twizzler paper [Bit+20a] describes gates as a way to perform IPC, and calls between distinct programs, but in the context of this thesis it is sufficient to think of them as a region of allowed memory access.

3.3 Flags

Currently, flags in capabilities are used to specify which hashing algorithm to use to form a message to be signed. We allow for multiple algorithms to be used to allow for backward capability when newer, more efficient hashing algorithms are created.

The flags inside a capability is a bitmask providing information about distinct feautures of that capability. Currently we only use them to mark what hashing algorithm was used to form the message for the signature, but there's plenty of bits left to use. We hope for future work to develop more expressive ways of using capabilities, i.e. Decentralized Information Flow Control, as specified in 6.1.

May be worth discussing delegations, in order to describe how they could be extended from caps (as framework ofc)

Chapter 4

Security Contexts

Threads

Security Contexts are objects that processes attach to in-order to inherit the permissions inside the context. The contexts store capabilities, allowing for userspace programs to add capabilities to contexts, and kernel space to efficiently search through them to determie whether a process has the permissions to perform a memory access.

4.1 Base

Since security contexts can be interacted with by the kernel and userspace, there needs to be a consistent definition that both parties can adhere to, which we define. Objects in Twizzler have a notion of a `Base` which defines an arbitrary block of data at the "bottom" of an object that is represented as a type in rust. We define the `Base` for a security context as follows:

```
struct SecCtxBase {
    map: Map<ObjID, Vec<CtxMapItem>>, /// A Map from ObjIDs to possible capabilities.
    masks: Map<ObjID, Mask>,          /// A map from ObjID's to masks.
    global_mask: Protections,         /// Global mask that applies to granted prots.
    flags: SecCtxFlags,               /// Flags specific to this security context.
}
```

4.1.1 Map

The map holds positions to Capabilities relevant to some target object, which the relevant security context implementations for kernel and userspace to parse security context objects. Implicitly, the kernel uses this map for lookup while the user interacts with this map to indicate the insertion, removal, or modification of a capability.

How is the map in memory?

(I know it's flat... may be worth discussing in the context of not storing virtual address pointers.

4.1.2 Masks

Masks act as a restraint on the permissions this context can provide for some targeted object. This allows for more expressive security policy, such as being able to quickly restrict permissions for an object, without having to remove a capability and recreating one with the dersired restricted permissions.

The global mask is quite similar to the masks mentioned above, except that it operates on permissions granted by the security context as a whole rather than a mask per distinct object id.

4.1.3 Flags

Flags is a bitmap allowing for a Security Context to have different properties. Currently, there is only one value, UNDETACHABLE, marking the security context as a jail of sorts, as once a process attaches to it, it won't be able to detach. This acts as a way to limit the transfer of information if a thread attaches to a sensitive object. Once a thread attaches to such a context, it is forced to end

its execution with the objects that context grants permission to. We also plan to utilize these flags in future works, as described in 6.1.

4.2 Enforcement

All enforcement happens inside the kernel, which has a seperate view into Security Contexts than userspace. The kernel keeps track of all security contexts that threads in Twizzler attach to, instantiating a cache inside each one. Additionally, a thread can attach to multiple security contexts, but can only utilize the permissions granted by one unless they switch [Bit+20a]. To manage these threads, the kernel assigns a Security Context Manager, which holds onto security context references that a thread has.

The enforcement of security policy in Twizzler happens on page fault when trying to access a new object [Bit+20a]. Upon fault, the kernel inspects the target object and identifies the default permissions of that object. Then the kernel checks if the currently active security context for the accessing thread has either cached or capabilities that provide permissions. If default permissions + the active context permissions arent enough to permit the access, the kernel then checks each of the inactive contexts to see if they have any relevant permissions. If there exists such permissions, then the kernel will switch the active context of that process to the previously inactive context where the permission was found. If it fails all of these, then the kernel terminates the process, citing inadequate permissions.

Since the security context can have a mask per object, while also having a global_mask to the protections it can grant, the kernel also takes this into account while determining if a process has the permissions for access.

The original Twizzler paper [Bit+20a], and the following security paper go into more detail about the philosophy behind why enforcement works this way, such as the performance benefits of letting programs access objects directly without kernel involvement, etc.

may be worth summarizing a few more bits here
Doesnt have to be super detailed or anything
but its better to have a few thing
tie together than and a paragraph with "etc"

(eg recovering posix semantics and why that's desireable, allowing for "contained" threads, ...)

Chapter 5

Results

5.1 Validation

The first test is a basic scenario as a check to make sure the system is behaving as intended, and a more expressive test to demonstrate the flexibility of the model. Eventually, I intend to work with my advisor and peers to form a proof of correctness for the security model, as well as empirical testing to demonstrate its rigidity.

5.1.1 Basic

TBA

5.2 Expressive

TBA

5.3 Micro Benchmarks

Additionally, we have microbenchmarks of core security operations in Twizzler. All benchmarks were run with a Ryzen 5 2600, with Twizzler virtualized in QEMU. Unfortunately I ran out of time to perform benchmarks on bare metal, but they should be the same, if not more, performant.

why do you expect this? Characterizing this difference could be described as future work

5.3.1 Kernel

There a couple of things we benchmark inside the kernel, including core cryptographic operations like signature generation and verification, as well as the total time it takes to verify a capability.

Benchmark	Time
Hashing (Sha256)	267.86 ns ±163 ns
Hashing (Blake3)	125.99 ns ±117 ns
Signature Generation (ECDSA)	199.90 µs ± 9.45µs
Signature Verification (ECDSA)	342.20 µs ± 6.28µs
Capability Verification (ECDSA, Blake3)	343.59 µs ± 5.32µs

Table 1: Collection of Kernel Benchmarking Results

is this with SIMD in kernel? may be worth discussing this nuance

how many times did you run the experiment and how were the stats calculated?

We see that signatures are vastly more expensive than hashing, on an order of 10^3, meaning that your choice of hashing algorithm doesn't affect the total time taken for the verification of a capability. It's also important to note that this cost of verifying a capability for access is done on the first-page fault, then the kernel uses caching to store the granted permissions and provides those on subsequent

could be interesting to compare sig verification costs as the amount of data to verify goes up

page faults into that object. In the future, I hope to measure the difference between a cached and uncached verification. Secondly, we only measure verification inside kernel space; as discussed in section 3, capability creation only takes place in user space.

5.3.2 UserSpace

In userspace, we benchmark keypair and capability creation, as these operations are core to creating a security policy.

Benchmark	Time
Capability Creation	$347.97\ \mu s \pm 5.78 \mu s$
Keypair Objects Creation	$651.69 \mu s \pm 187.90 \mu s$
Security Context Creation	$282.10 \mu s \pm 119.90 \mu s$

Table 2: Collection of UserSpace Benchmarking Results

Almost all the time spent in creating a capability is the cryptographic operations used to form its signature, which is why it's in the same ballpark as the signature creation we saw earlier.

The high standard deviation in Keypair objects and Security context creation happens from the unpredictable time it takes for the kernel to create an object on disk. The reason keypairs are almost 2x more expensive since they create two separate objects, one for the signing key, and one for the verifying key.

Chapter 6

Conclusion

So, this is more a summary which is good to have here but you'll want to have some conclusions -- e.g. what did you learn (eg about the cost of the operations)

In short we provide a general overview of the critical security components for security system in Twizzler, along with implementation details and desgin descisions. The evaluation programs show how security policy can be expressed and verifies that the kernel is enforcing as programmed. Lastly we go over microbenchmarks to show and explain the cost of these operations.

6.1 Future Works

maybe go into more detail here. There's a number of things that are discussed as future work throughout that could use a couple sentences each here.

In the future I hope to take the primitives created during my thesis, and apply them towards the implementation of Decentralized Information Flow Control, as described in [Kro+07], into the Twizzler security model. Additionally I would love to see how the current security model evolves once we start adding distributed computing support to Twizzler, as described in the orignal paper [Bit+20a].

6.2 Acknowledgements

I couldn't have done the work for this thesis and for Twizzler if it wasn't for the support I've recieved from my advisor Owen Arden and my technical mentor Daniel Bittman! I owe both of you so much, not just for the class credit but also for how much I've learned in this endeavor. Thanks guys!

Bibliography

[Bit+20] Bittman D, Alvaro P, Mehra P, Long DDE, Miller EL. Twizzler: a Data-Centric OS for Non-Volatile Memory. In:. 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association; 2020a, pp. 65–80.

[ZL09] Zhai G, Li Y. Analysis and Study of Security Mechanisms inside Linux Kernel. In:, 2009, pp. 58–61. https://doi.org/10.1109/SecTech.2008.17.

[Lev84] Levy HM. Capability-Based Computer Systems. USA: Butterworth-Heinemann; 1984.

[Bit+20] Bittman D, Alvaro P, Mehra P, Long DDE, Miller EL. Twizzler: a Data-Centric OS for Non-Volatile Memory. In:. 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association; 2020b, pp. 65–80.

[JMV01] Johnson D, Menezes A, Vanstone S. The Elliptic Curve Digital Signature Algorithm (ECDSA). Int J Inf Secur 2001;1:36–63. https://doi.org/10.1007/s102070100002.

[Kro+07] Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek MF, Kohler E, et al. Information flow control for standard OS abstractions. SIGOPS Oper Syst Rev 2007;41:321–34. https://doi.org/10.1145/1323293.1294293.