

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*.

Date of the public defence: Members of the Thesis Committee:

August 28, 2005

Dr. Peter Alvaro

Dr. Andi Quinn

Abstract

whatevea lowkey not even sure what to write

Contents

1 Introduction	2
1.1 Data Centric Operating Systems	2
1.2 Capability Based Security Systems	2
1.3 Our Contributions	2
2 Key Pairs	3
2.1 Abstraction	3
2.2 Compartmentalization	3
3 Capabilities	5
3.1 Signature	5
3.2 Gates	5
3.3 Flags	5
4 Security Contexts	6
4.1 Enforcement	6
4.2 Base	6
4.2.1 Map	6
4.2.2 Masks	6
4.2.3 Flags	7
5 Results	8
5.1 Validation	8
5.1.1 Basic	8
5.1.2 Expressive	8
5.2 Micro Benchmarks	8
5.2.1 Kernel	8
5.2.2 UserSpace	8
6 Conclusion	9
6.1 Future Works	9
Bibliography	10

Chapter 1

Introduction

In mainstream operating systems, security policy is enforced at runtime by a omniscient and all powerful kernel. 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 accesses must be done through the kernel, so why not perform security checks right along-side an access. However the enforcement of security policy starts getting complicated when we try to separete the access mechanisms from the kernel. This notion 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. Provide direct, kernel-free, access to data.
2. 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 our previous security model fails to operate here as, by defenition, the kernel cannot be infront of accesses 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 ACL’s 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 large seperation 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, allows for users to completely create and manage security policy while the kernel is left to enforce it. This paradigm allows for the 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, with comparsons to micro-benchmarks done with an older version of twizzler. Code can be found in this [Github tracking issue](#).

Chapter 2

Key Pairs

Key pairs in Twizzler are the representation of the cryptographic signing schemes used to create a signed capability, discussed in 3.1. We design the keypair objects to be agnostic towards what cryptographic schemes are underneath, allowing for the underlying algorithm to be changed [Bit+20a]. The keys themselves 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 security policy, while also being flexible enough to make basic policy easy.

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 now means that Alice now can create new keypairs, protected by 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 of hers. It forms an elegant solution for capability creation without the involvement of the kernel.

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 found with a private key size larger than 256 bytes, we would have to drop backwards compatibility. Sure this can be prevented by setting the maximum size to something larger, but that 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. You can also specify default protections for an object or create a capability with the signing key and any desired permissions.

The neat thing about this design is that you can use a single keypair in-order to use any arbitrary amount of objects. An example could be a collection of objects holding files for a class, and grouping all of them under the same key. In short, having this flexibility allows for a significant debloating of the filesystem, compared to creating a new keypair for every single object.

In planned future work , as we talk more about in we can investigate the This results in the possibility of finegrained access control to semantic groupings of objects.

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. A Capability is built up of the following fields.

```
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 inside the capability.
}
```

3.1 Signature

The signature inside is what determines the validity of this capability. The only possible signer of some capability is who ever has permissions to the signing key object, or the kernel. In this way, if the signer decides to make the signing key private to them, no other entity can administer this signature for this capability. 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 message being signed to form the signature is the bytes of each of the fields inside the capability being hashed. There is support for multiple hashing algorithms as described in 3.1.

3.2 Gates

3.3 Flags

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

There is also plenty of space left in the bitmap, allowing for future work to develop more expressive ways of using capabilities, such as planned future work to implement information flow control into the twizzler security system.

Chapter 4

Security Contexts

Security Contexts are objects that store capabilities, which processes can attach onto, inheriting the permissions granted by the capabilities that reside inside.

4.1 Enforcement

The enforcement of security policy in Twizzler happens on page fault when trying to access a new object [Bit+20a]. Then the kernel inspects the security contexts attached to the accessing process, looking up what capabilities those contexts hold and if they are applicable to the object being accessed. 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.

4.2 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.  
}
```

4.2.1 Map

The map holds positions to Capabilities relevant to some target object, which the relevant security context implementations for kernel and userspaces 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.

4.2.2 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 desired 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.

4.2.3 Flags

Chapter 5

Results

All testing was done in QEMU, with a Ryzen 5 2600 processor.

5.1 Validation

5.1.1 Basic

5.1.2 Expressive

5.2 Micro Benchmarks

5.2.1 Kernel

5.2.2 UserSpace

Chapter 6

Conclusion

6.1 Future Works

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

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>.