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

Chapter 1

Introduction

In mainstream operating systems, security policy is enforced at runtime by a omnicient 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 great since the kernel is always **there** and the only way to access anything through it. However the enforcement of security policy starts getting complicated when we try to seperate the access mechanisms from the kernel.

1.1 Data Centric Operating Systems

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

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.

1.2 Capability Based Security Systems

Capability based security systems utilize capabilities, a finegrained

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 [Bit+20] and an early draft of a Twizzler security paper. Additionally we evaluate these systems inside kernel and user space, with comparsions to micro-benchmarks done with an older version of twizzler.

Chapter 2

Key Pairs

Key pairs are the Signing and Verifying keys used to create Capabilities. We design the keypair objects to be agnostic towards what cryptographic schemes are underneath, allowing for the underlying algorithm to be changed [Bit+20]. The keys themselves are stored inside of objects, allowing for persistent or volatile storage depending on object specification. It also allows for the keys to fall under the security system, meaning that signing keys can be protected by a different keypair, forming this chain of trust.

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 dat arepresentation allows for any arbitrary alogrithm 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. This results in the possibility of finegrained access control to semantic groupings of objects.

An example could be a colletion of objects holding files for a class, and grouping all of them under the same key.

Now restricting access to that one key prevents the usage of that key to create any new 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

4.1 Map

Chapter 5

Results

Chapter 6

Conclusion

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; 2020, 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.