M10

Security

Executive Summary

The M10 Platform strives to deliver a high level of performance, availability, and security. To do this, M10 needs to mitigate several attack vectors while maintaining extremely high throughput.

Any system handling large amounts of money will naturally become a target for many different malicious actors, be it hostile nation-states, criminal syndicates, or an exceptionally bright sixteen-year-old.

Double Spend & Byzantine Consensus

One of the fundamental tenets of modern software design is high availability. M10 wants to achieve at least 99.999% uptime for all services in a more tangible form that means less than 5 minutes of downtime per year.

It is functionally impossible to satisfy these needs without redundancy. Redundancy requires having multiple nodes running in parallel, kept in exact sequence with each other. So if one of these nodes crashes, gets updated, or is destroyed through an act of god, the platform can keep running with no interruption.

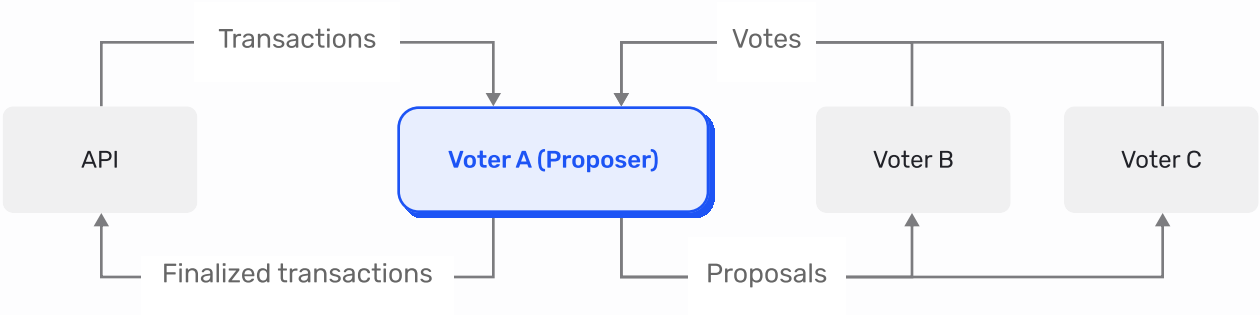
Though once you add multiple nodes into a system, you start to run into several problems.

Imagine that a user sends two transactions at virtually the same time. The user has a balance of $50. Imagine both transactions are for $50. If each node observes a separate transaction as arriving first, then both transactions could return as successful. In this case, the user effectively spent the same $50 twice. Double-spend can have disastrous consequences in the real world since it effectively creates money out of thin air.

It also means that each node is in a fundamentally different state, and there is no central shared ledger of transactions. This inconsistency can also arise if one of the nodes has been compromised and confirms invalid transactions.

M10 solves these issues through a highly performant algorithm called PaLa. PaLa is referred to in computer science as a Byzantine Fault Tolerant (BFT) consensus algorithm. Byzantine Fault Tolerance refers to the ability of the algorithm to withstand some compromised nodes without sacrificing integrity, or availability. Specifically, we are concerned with nodes that will "lie" to other nodes about their decisions (i.e., declaring ABC transaction ordering on one node and BAC on another). BFT algorithms are a hallmark of blockchain technology, often stigmatized as slow and environmentally damaging networks like Bitcoin. PaLa is a blockchain-based algorithm, but our implementation does not use Proof-of-Work, which is what causes many blockchains to waste electricity. The M10 platform is a private blockchain among trusted committee members; only these operator's servers can see the raw blockchain transactions, which guarantees privacy -- which traditional blockchains do not.

The core concept of PaLa is that each node votes on a "block" of transactions to decide if that ordering of transactions is valid. The nodes elect a "proposer," responsible for assembling new "blocks" and proposing them to the rest of the nodes. The other nodes can rotate the proposer if it starts misbehaving. PaLa is a particularly efficient algorithm, which allows us to reach high speed without compromising the core security and consistency principles of BFT.

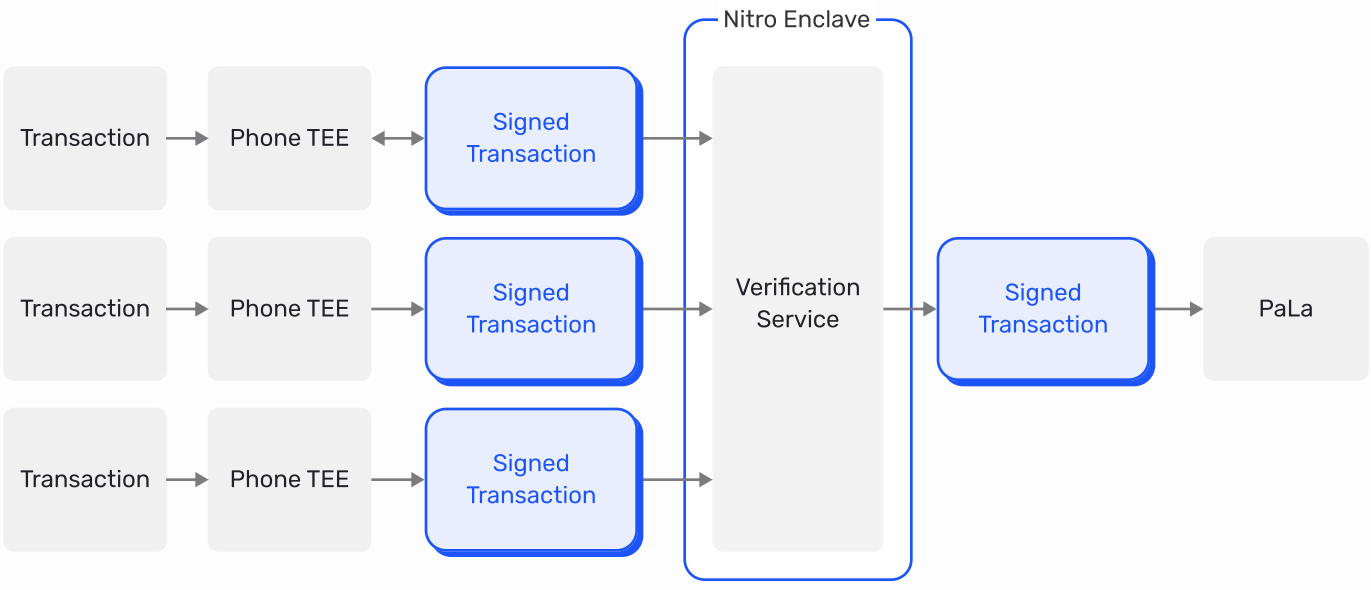


The core concept of PaLa

Authentication

Impersonation attacks are rampant in the traditional financial system. Hackers take over Twitter accounts of high-profile individuals, take out credit cards in others' names, and surreptitiously transfer funds out of bank accounts. Inadequate authentication systems cause many of these issues. Authentication is the practice of verifying that a user is whom they claim to be. Many systems use usernames and passwords to provide this guarantee; secure, difficult to guess passwords are often lost, though, so developers often implement a password recovery system.

These systems are ripe for exploitation since they essentially allow an attacker to gain control over a user's account. Username and password authentication schemes utilize a special "session token;" when the user presents the server with a valid username and password, the server generates a unique "session token." There are many ways to create and validate these tokens, but they all have the same fundamental issue: stealing the tokens. Since this token needs to be attached to every request, it is trivial to steal a session token. There are ways to limit the impact of a leaked token, but any leak can have disastrous consequences in a financial system. Plus, it is hard to disambiguate a truly stolen token from a user trying to defraud the system. Think of the number of users who post an embarrassing tweet, only to claim they were "hacked."



Authentication System

M10 uses an alternate authentication scheme called asymmetric cryptography, or public-key cryptography. In particular, M10 uses elliptic curve (EC) public-key cryptography. Public key cryptography works by generating two corresponding large random numbers: the public and private keys. The user generates a signature of a message using their private key. We can then verify that signature with their public key. You can not generate a private key from a public key, meaning that users can share their public key without revealing their private key. In the M10 system, private keys are often stored inside special hardware on a user's mobile device, called a trusted execution environment (TEE). These hardened components on the phone mean that even rogue software on a user's device doesn't have access to the key.

The problem with EC cryptography is that verification can be pretty slow. Above, we discussed our consensus model, which requires that every transaction be verified N times. Our system needs to run at 1M transactions per second (TPS). To get to this high level of throughput, we need to have multiple nodes verifying transactions simultaneously. But that leaves us vulnerable to a compromised node verifying invalid transactions. We solve this problem by running the verification service inside of a TEE.

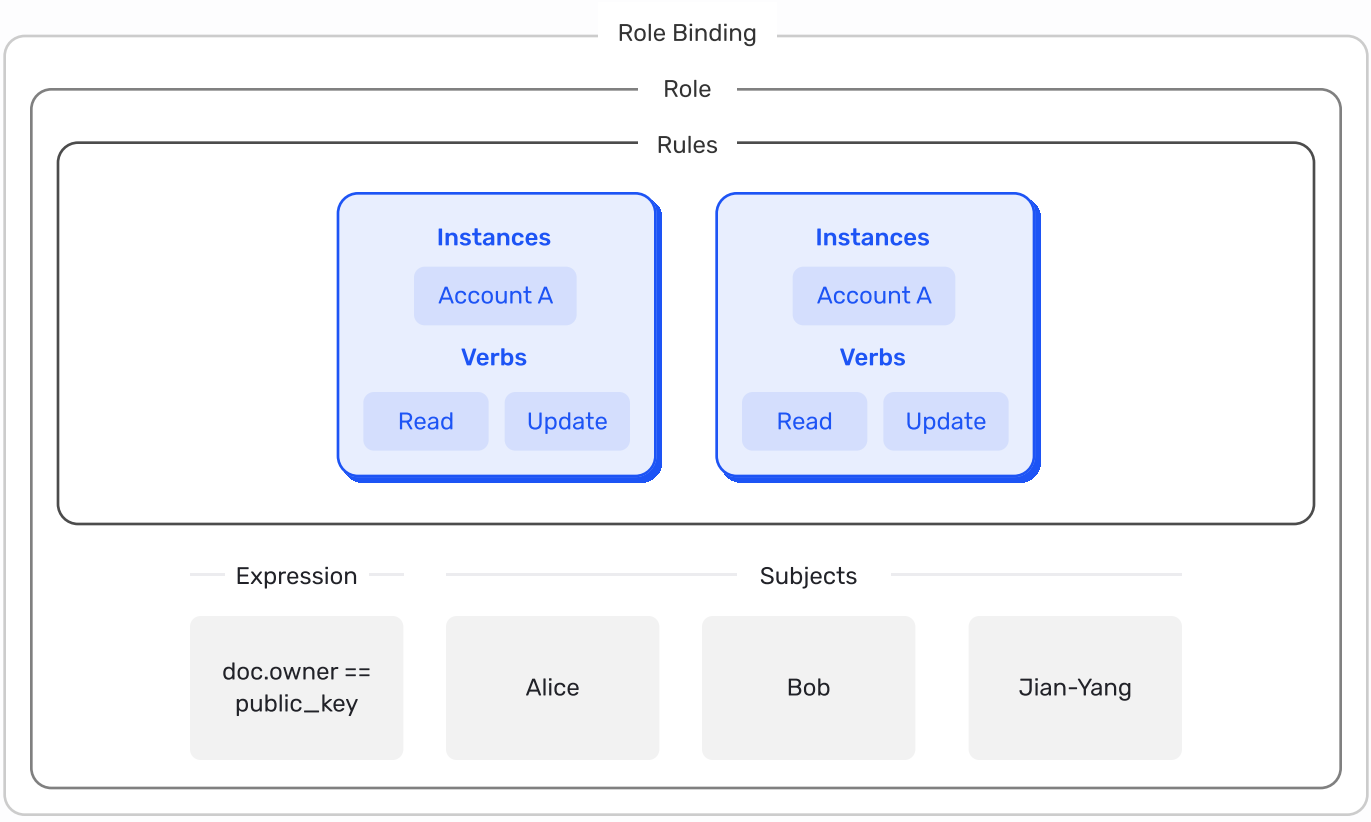
M10 supports two different types of TEEs: Intel SGX and AWS Nitro Enclaves. Both of these environments allow you to run software inside of a secure execution environment. Code running in an enclave (a TEE) is signed using an M10 specific private key. Programs running in the enclave signed with the M10 private key can encrypt, so it is only accessible by other programs signed by the M10 private key. M10 can then generate a private key that M10 signed programs in the enclave can only access. We then create a service that verifies blocks of transaction signatures and then resigns them with this trusted private key. M10 will only ever distribute valid copies of this verification service, which allows us to trust the verification results.

Authorization

Authentication is only one-half of the problem. It's still necessary to verify that a certain user (or, more specifically, their public key) should have access to an account. M10 uses a Role-Based Access Control (RBAC) system to handle this. RBAC works through Roles that grant certain permissions to resource types.

These permissions are things like "create", "update", "delete", "read", or "transact". For example, a user can be granted "read" permissions on an account. One can also specify specific resources to which a role applies. So you can grant a user transact permissions on a single account. Next are Role Bindings which grant a user the permissions of a Role. Role Bindings can be conditionally granted based on an "expression." An expression is a small check based on the context of an RBAC check. This might be something like checking that a user owns an account or that a transaction is below a certain number of dollars. Users can grant permissions on various resources they have permissions on. For example, if you can transact on an account, you can grant permissions so others can transact on that account.

RBAC is a highly expressive model for authorization that provides a large amount of flexibility and security.



Memory Safety

Around 70% of software vulnerabilities are caused by one thing: memory safety [1] [2]. Memory safety bugs are a class of software issue caused by misuse of memory. In traditional programming languages like C and C++, you get "pointers" to a location in memory. Pointers allow software programs to read data at the pointer's location. But what happens if you delete the data at that location and it is replaced by something else? The pointer will now return garbage data, potentially crashing your program or, worse, leaking it to someone else. A similar bug is called a "buffer overflow" attack. A buffer is a contiguous set of memory used to store data of some sort. You can imagine it has to have a starting location and a length. Many programs let the user pass in the size of there and did not perform extra verification that the buffer extended that long, causing bugs where users could read and write arbitrarily from the program's memory. This issue allowed data to leak, or worse yet, arbitrary code execution. Both of these represent serious vulnerabilities. Heartbleed [3] is one of the most famous examples of this type of vulnerability.

M10's backend services were written in the Rust programming language, which makes these sorts of bugs nearly impossible. Rust does this through a clever language construct called the borrow checker, which guarantees that you will only be able to access valid memory.

1 A proactive approach to more secure code – Microsoft Security Response Center

2 Memory safety - The Chromium Projects

3 The Heartbleed Bug

Content Trust

A common form of vulnerability is a supply-chain attack, where a backdoor or other nefarious behavior is surreptitiously added to a piece of software or an upstream dependency of that software. XcodeGhost [1] is a famous example of this where 128 million users were affected by software with exploits added to it.

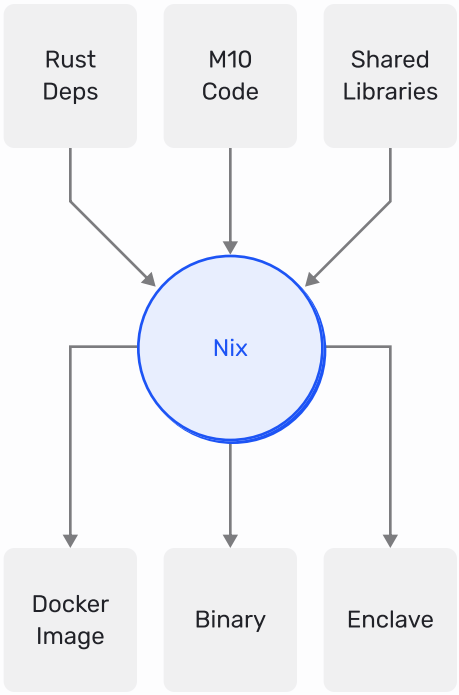
M10 uses multiple techniques to mitigate these risks. The Rust community takes security as a high priority; M10 utilize a community project called RustSec [2] that maintains a database of known vulnerabilities. Rust also confirms that the packages it is downloading are the same as a local file at build time. This signature checking helps prevent packages from being overwritten in the package repository.

1 A proactive approach to more secure code – Microsoft Security Response Center

2 RustSec Advisory Database

M10 also uses a system called Nix for reproducible builds. Reproducible builds mean that you can build the same software version on different computers and get identical binaries. Therefore you can build the software on multiple computers and verify that each binary is the same. If they are different, you know there has been a compromise of some sort. By doing this, we get a more significant deal of confidence that no one has compromised one of our build-machines.

Further, M10 signs all Docker images and enclave builds using a custom secure signing solution. The solution leverages M10's unique experience in enclaves with reproducible builds to form an extremely secure signing solution. Each Docker or Enclave image needs to be signed by a quorum of M10 developers; once that quorum has been reached, the Nitro enclave signs the image with the M10 private key.



Nix System

Infrastructure

M10 has built a secure platform, but any software is only as secure as the infrastructure that runs it. M10 uses AWS in production, and strictly follows best practices. Two factor authentication is required on all AWS accounts. Each account's access is limited, so if an account is compromised the damage they can do is limited. M10 uses up-to-date versions of Kubernetes, a system for managing software in a cloud environment, and Linux to prevent OS level vulnerabilities. To minimize the surface area an attacker can use to reach M10, incoming and outgoing traffic of the cloud environment is restricted. M10's engineers laptops and devices are all encrypted so if they are stolen hackers won't be able to use the keys on those laptops. Data in databases is encrypted at rest, so if AWS itself gets compromised there is a lessened risk of PII leaking. For internal services, M10 uses Hashicorp Vault to rotate secrets, and provide strong authentication guarantees.

Your Questions

This document is a high level description of the M10 security system. Therefore, many details have been omitted.

M10 will gladly answer any questions you may have to further deepen your knowledge about M10. Please contact us at info@M10.io.