sRPC

A Secure RPC Framework

6.828 Class Project Proposal

Jay Lang
MIT EECS
jaytlang@mit.edu

Ravi Rahman
MIT CSAIL
r_rahman@mit.edu

***Abstract*—Traditional remote procedure call (RPC) frameworks serve all RPC endpoints as one operating system user – a design that does not permit privilege separation nor follow the principle of least privileges. Unlike these frameworks, sRPC introduces a security-focused RPC architecture that supports discretionary and mandatory access control, spawns separate processes per each user, and benefits from the Unix file-system design and security primitives. We built a prototype implementation of the sRPC system in Python and evaluated its design with a security analysis and performance measurements. It successfully provides a simple interface for monolithic RPC servers to benefit from security best-practices; it limits the blast radius of remote code execution attacks; and its performance is within an order of magnitude of performance-optimized RPC frameworks. These initial results indicate that a security-focused RPC architecture can scale to real-world applications.**

I. Introduction

sRPC introduces a general purpose RPC framework built upon the Unix security model. Existing RPC frameworks require that RPC servers validate and authenticate requests, and only execute requests that the user is authorized to access. If unauthenticated users - or authenticated users with lesser privileges - could take control over the RPC server process, they could bypass RPC permission controls and access all resources the RPC server process can access uninhibited. Monolithic applications, where a single process serves multiple RPCs for users of multiple permission levels (e.g. both customers and site administrators), no longer obey the "principle of least privilege." The process will need to have permissions to serve the most privileged users. Should users be able to exploit security vulnerabilities in the application, they would be able to steal data, or worse, corrupt system state.

Unlike these existing frameworks, sRPC offers a new, security-focused approach for RPCs. Instead of having monolithic RPC server processes executing under operating system permissions for the most invasive of procedure calls, sRPC serves RPCs executing with proper permissions for the authenticated user. Through mandatory access control, a security group can specify which resources are available for each (user type, RPC endpoint) tuple. This design follows the principle of least privilege on a per-RPC basis, where each RPC handler process can access only the resources it needs to handle the request.

While sRPC focuses on security, its design also provides a scalable, easy-to-use interface. It extends upon popular RPC frameworks and supports RPCs written in any language. Section II describes existing RPC frameworks and Unix security models. Next, we present the sRPC architecture in 1. In IV, we detail our proposed implementation, and in V, we describe how we evaluate our system. Finally, in VI, we describe the impact of our work and techniques to further improve the system.

II. Related Works

A. RPC Frameworks

A popular RPC framework, gRPC [1], introduces a binary remote procedure call protocol and libraries in multiple programming languages. It uses protocol buffers (protobufs) [2] for request and response messages. It includes support for unary requests and responses, as well as streaming requests and responses. gRPC is popular because of its ease of use and support for many programming languages. Its design as a layer 7 HTTP/2 protocol enables compatibility with existing networking infrastructure, such as HTTP-based load balancers, and allows it to benefit from TLS encryption. However, this compatibility sacrifices performance with TCP head-of-line blocking, and sacrifices security by inheriting the permissions of the web server, which likely has access to resources not needed for serving the RPC.

While gRPC balances security, performance, and compatibility, many frameworks prioritize performance. One such framework, R2P2 [3], introduces a UDP-based transport for datacenter RPCs. Using UDP instead of TCP minimizes tail latency. To further improve performance, it decouples the client-server relationship and instead uses a middlebox for request-level load balancing. However, unlike gRPC, end-to-end encryption is impossible since there is never a client-server handshake; the design requires all connections to be in clear text. As such, it cannot be used over untrusted networks.

Unlike R2P2, other RCP frameworks focus on security. OKWS [4] offers a web server architecture that runs each RPC endpoint in its own process. Endpoint segmentation prevents the entire web server from crashing with just one buggy endpoint. In addition, should an endpoint have a security vulnerability, the operating system will confine the attack to the resources available in that endpoint; an attacker would not be able to access other endpoint processes. However, the design of OKWS does not offer process isolation by user. An attacker who is able to compromise one endpoint would be able to access resources available in that endpoint for all users. While an improvmeent over the traditional superuser process for the RPC server, it does not fully support the principle of least privileges.

B. Unix Security

Capsicum offers a lightweight sandboxing framework for Unix [5]. It provides fine-grained rights on file descriptors and limits which namespaces programs can access. Using `libcapscium` to define the set of permissions they need, applications then enter a sandbox, where they can access only the predefined methods. A process cannot exit the sandbox once in it; it can only become more restricted. This permission model can protect against vulnerability exploitation.

Security Enhanced Linux (SELinux) offers mandatory access control (MAC) for Linux systems [6]. Unlike discretionary access control (DAC), which is maintained by the end user, mandatory access control is managed centrally by a system administrator or developer. As such, MAC is generally regarded as more secure than DAC, as users cannot alter security attributes on all files they own at their own discretion. In addition, DAC policy rules are generally too coarse-grained to stop misbehaving applications running as a certain user from accessing sensitive files that belong to that user. SELinux offers a labeling system to construct a set of rules that specify which subjects can perform which operations on which objects, integrating Domain Type Enforcement (DTE) as well as Role-Based Access Control paradigms in conjunction with traditional MAC protections. Unlike Capsicum, SELinux does not require existing applications to be modified to run under SELinux systems, which is a boon for portability - though certain applications may be 'SELinux aware' within the bounds of their assigned context.

C. File-like I/O Over Networks

Plan 9 from Bell Labs [7] is a distributed operating system which utilizes a pervasive, application-layer file service protocol called 9P to abstract and multiplex remote resources - including disk storage, processor resources, and other networking devices - over the network to client terminals. The interface for file I/O Plan 9 uses is unique, employing 9P to perform almost all interaction between different components of the hybrid kernel. Though Plan 9's file-system oriented model of computing is not widely adopted at the OS level, the 9P back-end is utilized in systems such as Microsoft's Windows Subsystem for Linux and QEMU to perform lightweight transactions on file-like objects with implicit session management, and has been implemented in the Linux kernel.

As such, 9P heavily influences our custom RPC protocol, as it acts as an efficient alternative to UNIX sockets while allowing the use of regular read/write system calls to interact with remote systems.

D. Contributions

sRPC innovates on existing RPC frameworks by making the following key contributions:

- We utilize the Linux file name-space in order to craft and present a *RPC topology* to clients.
- sRPC endpoints map directly to socket files and named pipes, and accordingly support UNIX file abstractions, such as discretionary and mandatory access control.
- sRPC incorporates privilege separation on a per-user basis. It serves each user's RPCs from separate processes.
- The design of sRPC is resource efficient and supports asynchronous requests and parallel request handling.

Rather than present a single flat list of services, servers organize RPC endpoints in a nested, hierarchical, file-system. This treatment of endpoints is inherently easy to understand, provides strong organization, and natively inherits Unix file permissions and file operations. When a client connects to the server, it inherits this hierarchy and may interact with it via a 9P-like file service protocol - and the sRPC daemon marshals requests running as that user, in that user's security context.

All accesses to these pseudo-files on the server side are mediated through a mandatory and discretionary access control layer - the sRPC daemon. The MAC system in play may implement Domain Type Enforcement (DTE) and Role-Based Access Control (RBAC), along with potentially more complex schemes (e.g. multi-category security). Because all RPCs pass through server-side socket files, the underlying operating system is responsible for enforcing the security policy, and sRPC doesn't have to be aware of how MAC is being utilized (if at all) to serve its intended purpose. This approach is highly configurable while retaining the intuitive, file-system interface, and meshes well with existing MAC implementations such as SELinux.

While sRPC focuses on security, its design does not neglect performance. At the network level, it allows multiple users and mount points to share the same underlying TLS connection to minimize TLS overhead. On the server side, instead of requiring a new Unix process for each request, it reuses processes for the same user. This user-level sharing does not compromise security while avoiding the Unix process and system call overhead. Because RPCs are served through socket files, multiple server-side handlers can respond to the same endpoints to support per-user-endpoint parallelization.

III. System Design

A. Network Layer

sRPC uses TLS sockets for network communication. TLS provides strong encryption, handles session management, and guarantees ordering of request and response data through underlying TCP. sRPC uses certificate pinning to validate the server. The server must present a TLS key pair that the client accepts.

We designed a simple, custom message header for our application layer protocol to be passed over TLS connections, with data organized in the following order: a 4-byte integer message type identifier, a 8-byte unsigned integer sequence identifier, and a 8-byte unsigned integer message length (see figure 2). The variable length message follows this header. For simplicity, we encoded all messages in JSON for transmission over the wire.

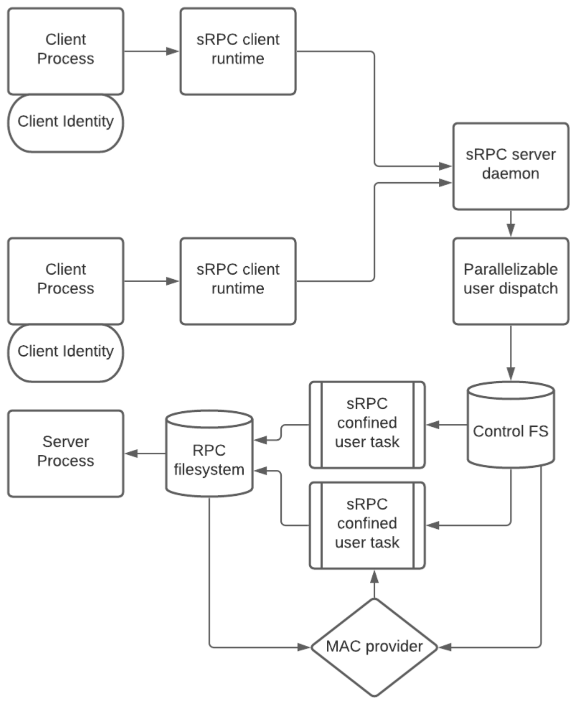

Fig. 1. The system architecture of sRPC, from client and server perspectives

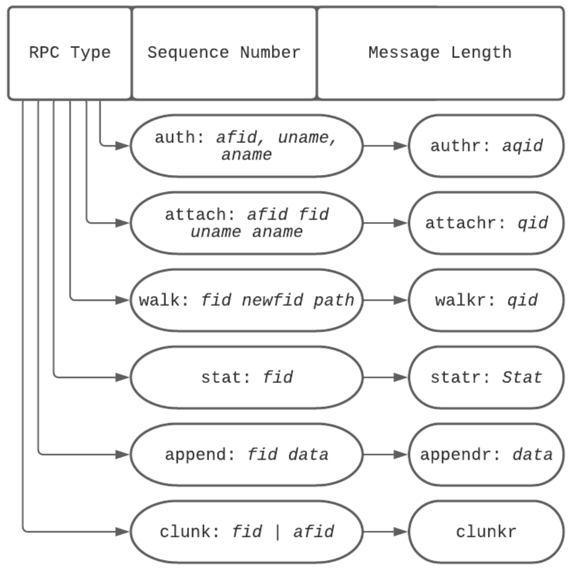

Fig. 2. The sRPC network protocol, including the message header - as well as built in remote procedure calls and their server response counterparts.

B. Application Layer

sRPC utilizes a custom network protocol for calling and responding to RPCs within the server's RPC hierarchy, modeled after the 9P file service protocol. In addition to the sequence number and message length passed in the protocol header, we also specify one of several RPC types, or commands the RPC server is able to respond to. As RPCs are presented as a hierarchy to the client, these commands resemble file-like operations one might perform on a local hard drive. A summary of these commands, along with parameters and server responses, can be found in figure 2.

Many of these commands utilize the notion of a client file identifier, or `fid`. Each `fid` acts as an immutable pointer to a server-side query identifier, or `qid`. These `qid`s map one-to-one to RPC file-like endpoints resident in the server filesystem, which are permissioned properly for authenticated clients to use and which the server can open, read, and write to. The `fid` is represented as a unique integer (much like a file descriptor) *of the client's choosing*, which is registered with the server through one of two calls:

1) `attach`: This command allows the client to map a new `fid` to a path within the server RPC hierarchy. This command takes four parameters: an identifier for the authentication context (an `afid`, which we will discuss later), a `fid` the client would like to point at the attach point, the username to attach as, and the attach point itself. Provided the authentication context is valid and matches the username / attach point pair passed, the server asynchronously clones the RPC tree beginning at the attach point, ensuring that file attributes are also copied. With the RPC hierarchy ready for client interaction, the client's `fid` of choice is mapped to the `qid` referencing the attach point, and this `qid` is returned to the user for reference.

2) `walk`: This command allows the user to map a new `fid` relative to an existing `fid`, hence descending the directory tree. Provided the relative pathname as well as the old and new `fid`s, the new `fid` is registered, pointing at the `qid` corresponding to the relative pathname. The associated `qid` is returned to the client, while the old `fid` is left intact.

Because `fid`s are immutable, and a new `fid`s can only be created via the above two commands, the ancestry of each `fid` is well defined. Thus, `fid`s can be used to perform implicit session management, ensuring that multiple user's interactions with a server may be securely multiplexed over a single connection with no additional implementation hurdle.

A `fid`, once properly mapped to a `qid`, may be interacted with through the following commands:

3) `stat`: Given a `fid`, this command returns a `Stat` structure containing useful information about the file this `fid` references, including whether it is a directory, any children it might have, its `qid`, and other attributes such as ownership and security context.

4) `clunk`: This command unregisters a `fid` with the server, leaving it free for reassignment via `attach` or `walk`.

5) `append`: This command issues a write, immediately followed by a blocking read, to a given `fid` in a single network traversal. Keeping up the abstraction of RPCs as files, this is equivalent to calling the RPC (by writing arguments to the file-like endpoint) and having the server asynchronously return the results (via the server writing back to the client).

C. Client Authentication

After establishing the TLS connection, authentication may be performed in a number of ways, and the sRPC API leaves this to the server through the `afid` abstraction. Much like a normal `fid`, the `afid` references a file-like object on disk, but the sole purpose of this file is to be used as a two-way channel for the client to authenticate itself to the server. Depending on the MAC implementation, a common way of accomplishing this is through requiring the client to `append` its credentials, and verifying them against the Linux password store (through PAM), and in turn map Linux users to a set of security roles. Regardless, after authentication is performed, the server can mark the `afid` as accepted or rejected, which determines whether a subsequent `attach` may take place.

An `afid` is established through an `auth` command. This command takes three parameters: an `afid`, the username to authenticate with, and a future attach point. The server creates an entry in a global authentication table for this identifier. This entry is marked as unauthenticated, and is made available to the server for communication (or shimmed with an authentication layer inside sRPC itself). To successfully authenticate, the client must call `append` to supply authorization credentials, and then supply an `attach` with a matching `afid`, username, and attach point.

sRPC is capable of using an `afid` multiple times to attach to the RPC hierarchy, including to subdirectories of the originally requested attach point.

D. Security management

After an incoming connection is validated and initiated through `auth/attach`, the sRPC daemon lazily allocates and drops into a new worker process, running within the authenticated user's security context. This immediately restricts which RPCs are available to the client, as RPCs are permissioned appropriately in the filesystem itself, and calling each RPC through `append` constitutes writing to the object in the filesystem. Delegating to the filesystem-based MAC provider rather than reimplementing this functionality is a novel approach, and means that the server handler code does not need to be aware of the specific security system used. The object manager will transparently control reads and writes according to the assigned security context, so that the server cannot make changes on behalf of a malicious client - making sRPC extremely portable across MAC and DAC approaches.

IV. Implementation

We implemented sRPC as an Python package. All functions are non-blocking with the `asyncio` framework, which provides easy, thread-safe concurrency. Asyncio provides built-in interfaces for working with TLS sockets, Unix domain sockets, and named pipes. Because Python does not easily support inter-process communication, we used a control filesystem composed of UNIX domain sockets to communicate between the root server process and permissioned user work processes. The server interacts with sRPC through named pipes, which act like clones of the RPC filesystem and present with the same permissions as that hierarchy.

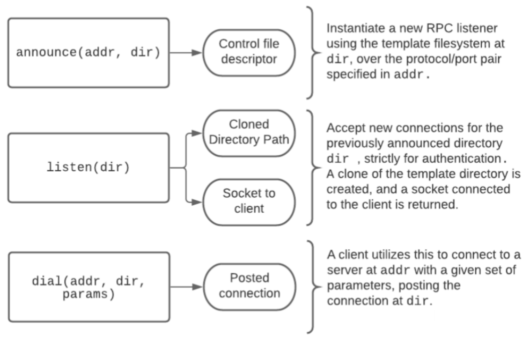
announce(addr, dir)
Control file descriptor
Instantiate a new RPC listener using the template filesystem at dir, over the protocol/port pair specified in addr.
listen(dir)
Cloned Directory Path
Socket to client
Accept new connections for the previously announced directory dir , strictly for authentication. A clone of the template directory is created, and a socket connected to the client is returned.
dial(addr, dir, params)
Posted connection
A client utilizes this to connect to a server at addr with a given set of parameters, posting the connection at dir.

Fig. 3. The sRPC API, from both server and client perspectives

User processes are created lazily, specifically, when an authenticated user attempts to `walk`, `stat`, or `append`. To create this process, the server uses Python's `multiprocessing` module to create a new Unix process. The new process calls `os.setuid` and `os.setgid` to limit its permissions, sets the proper security context for a given user, and then binds to the socket file created by the root process. The root server then stores this control socket in a table (organized by the user), so it can re-use it whenever the user calls an RPC, regardless of which connection one may be calling from.

When a command is invoked, the root server asynchronously proxies requests and responses between the appropriate TLS connection and user process socket file. The user process then attempts to write the request to the specific RPC pipe. Should the user not have permissions to call an RPC, then the write will fail, and this error will be propagated back to the client. A server handler is responsible for monitoring these pipes to respond to requests. When a response is written back on the response pipe, the user process will write this response back to the socket file, which the root server can then propagate back to the client.

The sRPC package implements a simple API that mimicks a UNIX-like listen call for the easy implementation of servers based around sRPC, as well as an `announce` function which prepares such a server for listening. On the client side, we've specified a complementary `dial` function, which allows connection to an sRPC server. sRPC is designed to allow the posting of this connection as a named pipe set in the filesystem, in conjunction with presenting it as a typical socket file descriptor.

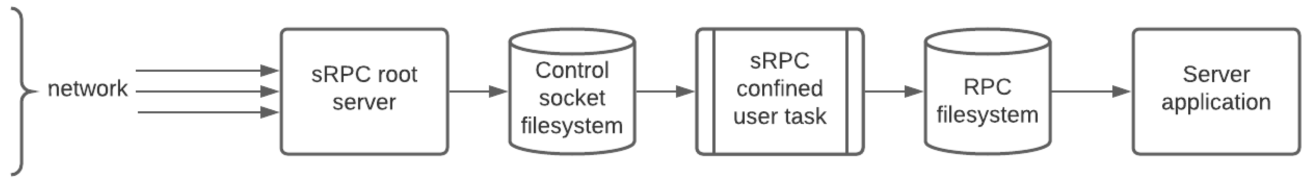

Fig. 4. The sRPC daemon's critical path when processing requests. Access to the filesystem can be easily parallelized, allowing for potentially significant performance optimizations.

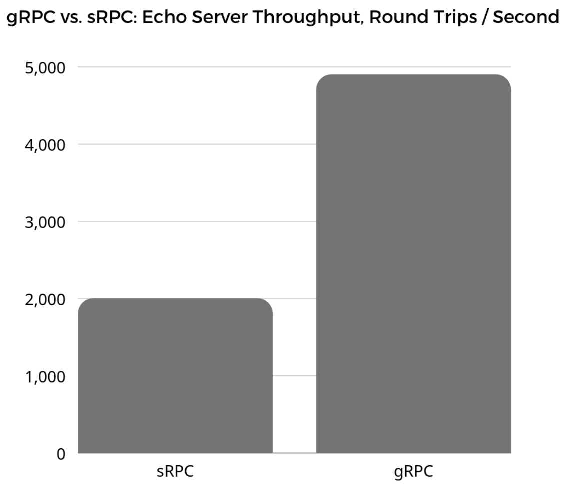

Fig. 5. sRPC performance as compared to gRPC, both running a simple echo server utilizing their respective Python 3 bindings.

V. Evaluation Methods and Metrics

We evaluate our system for security via a qualitative analysis, and as a secondary measure evaluate performance via quantitative and qualitative analysis.

A. *Qualitative Security Analysis*

1) Privilege Separation: Privilege separation refers to having different parts of the program execute with only the privileges that each part needs to operate. Unlike a traditional RPC server which serves all RPCs as a superuser (for simplicity and performance), sRPC separates the RPC handler from the authentication and routing layers. This design ensures that the RPC handler does not have access to the resources needed to complete other RPCs.

2) Remote Code Execution: Remote code execution, when an attacker is able to run unauthorized code on a remote machine, can occur with due to poorly written programs (such as with memory violations) or bugs in the run-time library. In a traditional RPC framework, remote code execution could overtake the RPC server. The attacker could intercept all requests and access any resource that the RPC server has access to. In sRPC, requests are handled in operating system processes running as the user specified in the request. Should there be a vulnerability in the RPC handler, remote code execution would still be possible, but because of privilege separation, the operating system would limit the resources accessible to the rouge process. In combination with mandatory access control and proper permissions on RPC files, the attacker would not be able to access resources designated for other users or RPC endpoints This permission system significantly limits the attack surface,

B. *Permissions Management*

Traditional RPC systems require multiple, duplicated levels of permission management. For example, web RPC frameworks rely on operating systems to manage permissions for the RPC server superuser and a separate database to store user authentication credentials. System administrators must ensure that the RPC server has operating system permissions to access all necessary resources (e.g. network, disk) for the users that the RPC server handles. These overlapping permission systems can lead to configuration mismatches. Unlike this architecture, sRPC only uses operating system permissions. Users and processes alike have permissions managed by the underlying operating system, so there is a single source of truth for access control.

C. *Quantitative Performance Evaluation*

Though it is not our primary evaluation metric, we test sRPC's performance quantitatively by stressing its critical path: single threaded, rapid request/response sequences. Consisting of repeated `append` requests on an existing connection, this path (shown in Figure 4) requires multiple passes through the host filesystem. We implement a simple sRPC echo client and server, and contrast its performance with a standard gRPC echo server, written using gRPC's Python to C bindings and utilizing protobufs.

Surprisingly, sRPC remains competitive with the gRPC implementation, achieving a throughput of 2,000 echo RPCs processed per second as opposed to gRPC's 4,900. This is well within an order of magnitude, and is especially auspicious as this implementation of sRPC is single-threaded: each stage of the critical path, delineated by filesystem accesses, is designed to be easily parallelizable. Performing this extra step could increase sRPC's performance even further, potentially in addition to reimplementing sRPC in C with shared memory instead of a secondary socket filesystem.

To address context switching overhead between potentially many user processes, only one new process is created per security context, even if multiple clients are connected over this same context. This single thread in turn will iterate over all connections in its context - thus requiring $O(context)$ threads rather than $O(connections)$ threads.

VI. Conclusion and Future Work

sRPC introduces a security-focused RPC framework that supports user-level privilege separation and mandatory and discretionary access control built into Unix systems. In addition to its secure design, its architecture is designed to minimize operating system and process overhead while supporting asynchronous requests and parallel request handling. Our prototype implementation demonstrated that the performance overhead from the strong privilege separation of sRPC is within an order-of-magnitude of leading RPC frameworks. To further enhance the performance of sRPC, the framework can be rewritten in C/C++, and inter-process sockets can be replaced with shared memory pages. These improvements would not compromise sRPC's secure design and further demonstrate that security does not come at a significant cost over performance.

References

[1] T. L. Foundation, "grpc: A high-performance, open source universal rpc framework," 2020. [Online]. Available: https://grpc.io/

[2] Google, "Protocol buffers." [Online]. Available: https://developers.google.com/protocol-buffers

[3] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2p2: Making rpcs first-class datacenter citizens," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 863–880.

[4] M. N. Krohn, "Building secure high-performance web services with okws." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 185–198.

[5] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix." in *USENIX Security Symposium*, vol. 46, 2010, p. 2.

[6] RedHat, "What is selinux?" [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux

[7] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Computing systems*, vol. 8, no. 3, pp. 221–254, 1995.