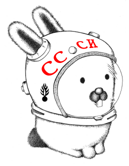
Dr Glendarme or: How I Learned to Stop Kerberos and Love Factotum

Edouard Klein[1]
Guillaume Gette[2]

[1] Beaver Labs, Paris, France
[2] Ecole Polytechnique, Palaiseau, France



ABSTRACT

Authenticating users and securing resources in a distributed system is hard because identity, ownership and permissions must be kept in sync across all machines in a domain, despite being intrinsically local properties handled by the various hosts' kernels. Yet, this is a necessary first step in piecing together a local-looking virtual system out of multiple remote resources, which in turn allows one to remove all security and networking concerns from application source code, improving both security and ease of development. We augmented factotum with the Guillou-Quisquater protocol and created a userspace server, a PAM module and a NSS configuration option which, with the already existing plan9port project, effectively backport large parts of Plan 9's distributed security model back to Linux. Compared to existing solutions such as Kerberos, our setup is trivial to install and administrate and now, one can develop a set of programs as if they would be run locally, but have them run as processes communicating transparently across multiple computers sharing a consistent access-control policy.

1. Introduction

Securely sharing resources between users on a monolithic, powerful mainframe has been made possible by, among others, the UNIX operating system and its descendants. Fine-grained access control is handled by the kernel on the basis of a user's identity and groups and of a processes' and files' owners, groups and permissions.

Our IT infrastructure is made up of a few overpowered servers and of many powerful workstations and laptops (as is most small to midsize institutions' infrastructure nowadays). We wish our users could compose their computing environments out of bits and pieces of all the machines on our network, seamlessly and securely sharing resources as if working on one big virtual mainframe.

We first got by thanks to a progressively heavier use of SSH, which appealed to us with its rock-solid security, its ability to tunnel any other network protocol and its ease of deployment, configuration and use. We nevertheless quickly considered a switch to another model because of growing pains such as:

Key management: because in a truly distributed system, every machine is liable to host both client and server processes, every machine had to know every other host's key fingerprint and every user's public key.

User management: user accounts and groups had to be kept in sync on all devices.

`uid` **mismatch:** the same user would have a different `uid` on different workstations.

Meaninglessness of port numbers: our heavy use of ssh tunneling was made more difficult as we had to avoid collisions and could not tell at a glance which ports were free and which services were being tunneled where.

LDAP and Kerberos are obvious choices to alleviate at least the first three pains points we felt with SSH. We did not, however, make the leap because of the difficulty of setup and administration and because of Kerberos' inherent reliance on sharing secrets with the application code, which is much more likely to leak those secret than code specifically developed for this purpose such as `ssh-agent` or `factotum`. (see subsection 5.6).

Instead, we need a model that would let us keep the two core attributes that dictated the way we had developed and deployed our application software so far:

Network Transparency: A process gets the impression that all resources it accesses are located on the same machine as itself. No application code need to care about connecting to a remote machine or receiving connections from anywhere but localhost.

Security Agnosticism: A server process runs, on the remote machine, owned by the user who will make all subsequent requests to it from the client machine. Therefore, a server process does not need to authenticate a user nor to check whether a request is legitimate. This job is delegated to ssh (the authentication part) and to the server host's kernel (the access control part).

We turned to the Plan 9 operating system (Pike et al., 1995), whose elegant security model (Cox et al., 2002) is exactly what we need: it relies on a secure piece of software called `factotum` to negotiate authentication between the server process and the client process. Those two processes only have to forward messages between the user's `factotum` and the server host's `factotum`, with no care for the meaning of those messages, thus ensuring that implementing new authentication protocols is simply a matter of upgrading `factotum`. This is *Security Agnosticism*.

Network transparency on Plan 9 comes from the ubiquitous use of the 9P protocol (Pike et al., 2019), which allow any remote resource to be mounted in the local namespace as a part of the file system.

As we can dream of neither porting Plan 9 to our diverse hardware nor porting our application software to Plan 9, we elected instead to backport parts of Plan 9's security model to Linux. This paper is a description of our successful, yet still early, work to this end:

vrs (subsection 2.3) We created a privileged daemon called `vrs` whose role is to start server processes owned by authenticated users. It ensures the server processes can be *Security agnostic* and runs amidst the already existing `Plan 9 from user space (plan9port)` tools (Cox et al., 2018) and `v9fs` (Van Hensbergen and Minnich, 2005) (subsection 2.2), which take care of all the *Network Transparency* of our setup.

Pluggable Authentication Modules (PAM) Module (subsection 3.1) We created a PAM module that handles the conversation between a client's `factotum` and the host's trusted `factotum`. The choice of which `factotum` to trust is therefore part of PAM's configuration, where one would expect to find it on Linux. This PAM module also allows any application to authenticate users with `factotum`, although we discourage application code from handling security at all.

Name Service Switch (NSS) (subsection 3.3) We modified the `factotum` from `Plan 9 from user space` to serve a `passwd` and `groups` file. Once mounted, these files can be pointed to in the NSS configuration, allowing all machines in the network to share the same user and group database.

Guillou-Quisquater (section 4) To avoid relying on shared secrets, we were able to add support for the Guillou-Quisquater zero-knowledge interactive proof (Guillou and Quisquater, 1988) with no particular effort thanks to the clean design of `factotum`.

2. Network-Transparent, Security-Agnostic servers and clients

2.1. Illustrative example

Figure 1 shows a typical example of what we strive to achieve. The network-wide user `alice` owns the machine `Atlantis`, but she wants to have some files on `bob`'s machine `Bermuda`.

`alice`'s process `foo` should have to care about neither the fact that the files it accesses are not actually on `Atlantis` (*network transparency*) nor whether `alice`, its owner, has any rights to those files (*security agnosticism*): the kernel should block any illegal action.

It is understood that if `bob` goes rogue, he can access `alice`'s files. `bob` is the owner of all of `Bermuda`'s resources, and if she wants to use those resources, `alice` must trust `bob`.

Our example deals with sharing actual files located on a hard drive spinning inside `Bermuda`. The 9P protocol (Pike et al., 1992, 2019) can intuitively be seen as mapping filesystem operations (`open()`,`write()`, etc.) to network messages whose names begin with `T` for the request and `R` for the response (`Topen/Ropen`, `Twrite/Rwrite`, etc.).

Via 9P, Plan 9's design allow most, if not all, resources to be presented as filesystems and thus mounted locally from a remote host: keyboard, mouse, and screen, network connections, and even debugger access to a remote process. As Pike et al. (1992) put it:

> files in Plan 9 are similar to objects, except that files are already provided with naming, access, and protection methods that must be created afresh for objects.

However, even today, twenty one years later, no widely used operating system can expose its host's resources in such a unified way. In our real life use-cases, most of the software our users need expose their functionality as HTTP endpoints speaking JSON or XML.

Nevertheless, we contend that despite its seemingly marginal adoption in the world of linux user-facing applications, 9P is still a valuable target :

- securely sharing actual files is still problematic (subsection 5.4),
- virtual file systems are slowly making their way into the collective mindset thanks to FUSE (Szeredi, 2019) (subsection 5.5),
- we are working on tunneling HTTP over 9P (subsection 5.3),
- 9P enjoys very good client support on Linux thanks to `v9fs` (Van Hensbergen and Minnich, 2005) and its use in mainstream applications such as QEMU (Jujjuri et al., 2010).

This example will let us explain the role of the existing plan9port tools in the *network transparency* of our setup (subsection 2.2) and explain how our new tool `vrs` is the missing piece in charge of the *security agnosticism* (subsection 2.3). We invite the reader to follow these explanations along with Figure 1 to Figure 6, which use the following conventions:

a host is a rounded corner box with its name in the upper part,

a process is a circle inside a host

a named UNIX socket is a diamond shape tied to a host,

a TCP port is an ellipsis tied to a host,

the kernel is the box on top of the host, while

the filesystem is the box below.

The color illustrates who owns what. `alice` is blue while `bob` is red.

A dashed arrow denotes a virtual relationship: one that is made of smaller components.

A solid arrow denotes a direct relationship, with no intermediary party (e.g. it denotes a system call like `listen`).

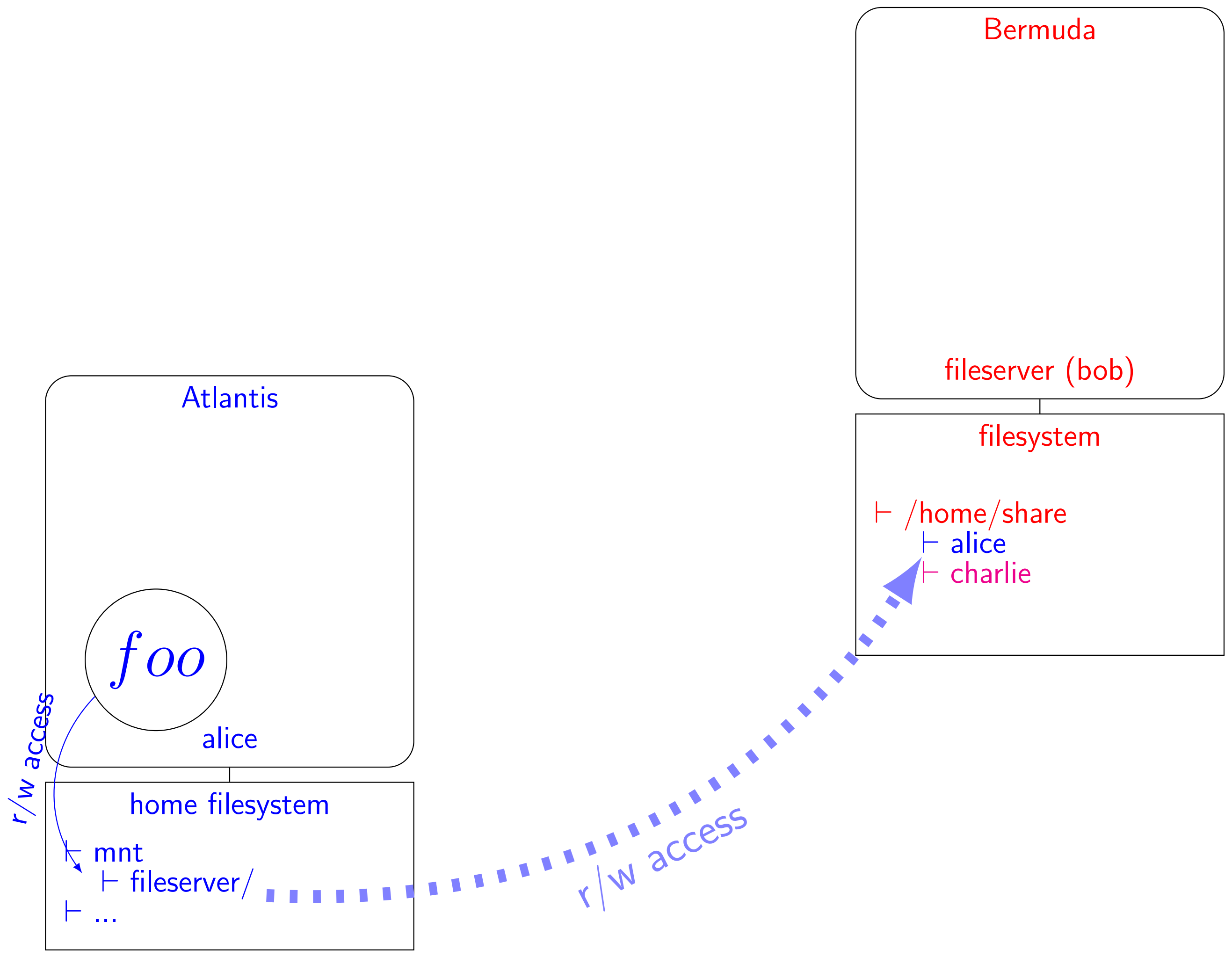


Figure 1: On `alice`'s machine `Atlantis`, a process `foo` owned by `alice` accesses `alice`'s files on `bob`'s machine `Bermuda` as if those files were on `Atlantis`. `Bermuda`'s kernel offers the same protection as `Atlantis`'s kernel would.

2.2. Network Transparency with `plan9port`

2.2.1. u9fs owned by alice

We need to translate what `foo` does to `Atlantis`' filesystem into similar actions on `Bermuda`'s filesystem. Therefore there must exist a process on `Bermuda` tasked with receiving 9P messages and translating them into system calls impacting `Bermuda`'s file system.

The existing `u9fs` fileserver does exactly that. It was designed to be *network-transparent*, as it reads and writes 9P on its standard input and output. In subsection 2.3 we will explain how the `u9fs` process is also *security agnostic*, because it is owned by `alice`.

This means that `Atlantis`' and `Bermuda`'s kernels must agree on who `alice` is, including her numeric `uid`[1], groups, and their numeric `gid`. In lieu of LDAP, we make both kernels refer to our modified `factotum` running on a central authentication server (see subsection 3.3). These components are drawn on Figure 2.

2.2.2. A factotum dialog

Before `u9fs` can be started on `Bermuda` under her identity, `alice` must authenticate on `Bermuda`. Following the Plan 9 security model (Cox et al., 2002), this is just a matter of having `alice`'s `factotum` talk with the `factotum` `Bermuda` sees as authoritative.

`factotum`'s communication and authentication protocols do not require any intermediary to interpret the messages. Any application intermediary only forwards messages between a `factotum` client and a `factotum` server to perform the authentication. Thus, `alice` just needs to run a client instance of `factotum` to authenticate on Bermuda, see Figure 3

2.2.3. v9fs and srv

To translate `foo`'s system calls into 9p messages, we use the kernel layer `v9fs` (Van Hensbergen and Minnich, 2005), via `plan9port`'s `mount` command.

`v9fs` intercepts system calls made by `foo` to access the file system, translates them in 9P, then writes the 9P messages to a socket. Which socket to connect to is decided when invoking the mount command. Here Figure 4 shows a named UNIX socket on `Atlantis`.

The 9P response messages read by v9fs from the socket determine the system calls outcomes, with `foo` being unaware of all this machinery.

Listening on the socket is the `plan9port` program `srv`. It first sends the Tauth message to the server, then relays messages back and forth between the server and `alice`'s factotum on `Atlantis`, until `alice`'s `factotum` tells it to stop.

In a second phase -if the authentication succeeded- `srv` creates the socket and simply forwards 9P messages back and forth between the server and whoever (here, `v9fs`) connects to the UNIX named socket, as shown in Figure 4.

2.2.4. 9pserve

The TCP socket on `Bermuda` that `Atlantis`' `srv` connects to is created by the `plan9port` utility `9pserve`. Its man page explains:

> [...] the Plan 9 kernel multiplexes the potentially many processes accessing the server into a single 9P conversation. The user-level server need not concern itself with how many processes are accessing it or with cleaning up after a process when it exits unexpectedly. On Unix, 9pserve takes the place of the Plan 9 kernel, multiplexing clients onto a single server conversation and cleaning up after clients when they hang up unexpectedly.

This 'single server conversation' can not be directly forwarded to `alice`'s `u9fs` on `Bermuda`. Other clients, such as one owned by `charlie`, for example, may be connected to the same port, too. `charlie`'s messages need to be forwarded to `charlie`'s instance of `u9fs` on `Bermuda`. Furthermore, `9pserve` is unable to authenticate a connection. It expects the server to handle Tauth messages. See Figure 5

[1]Plan 9 uses actual strings instead of numbers

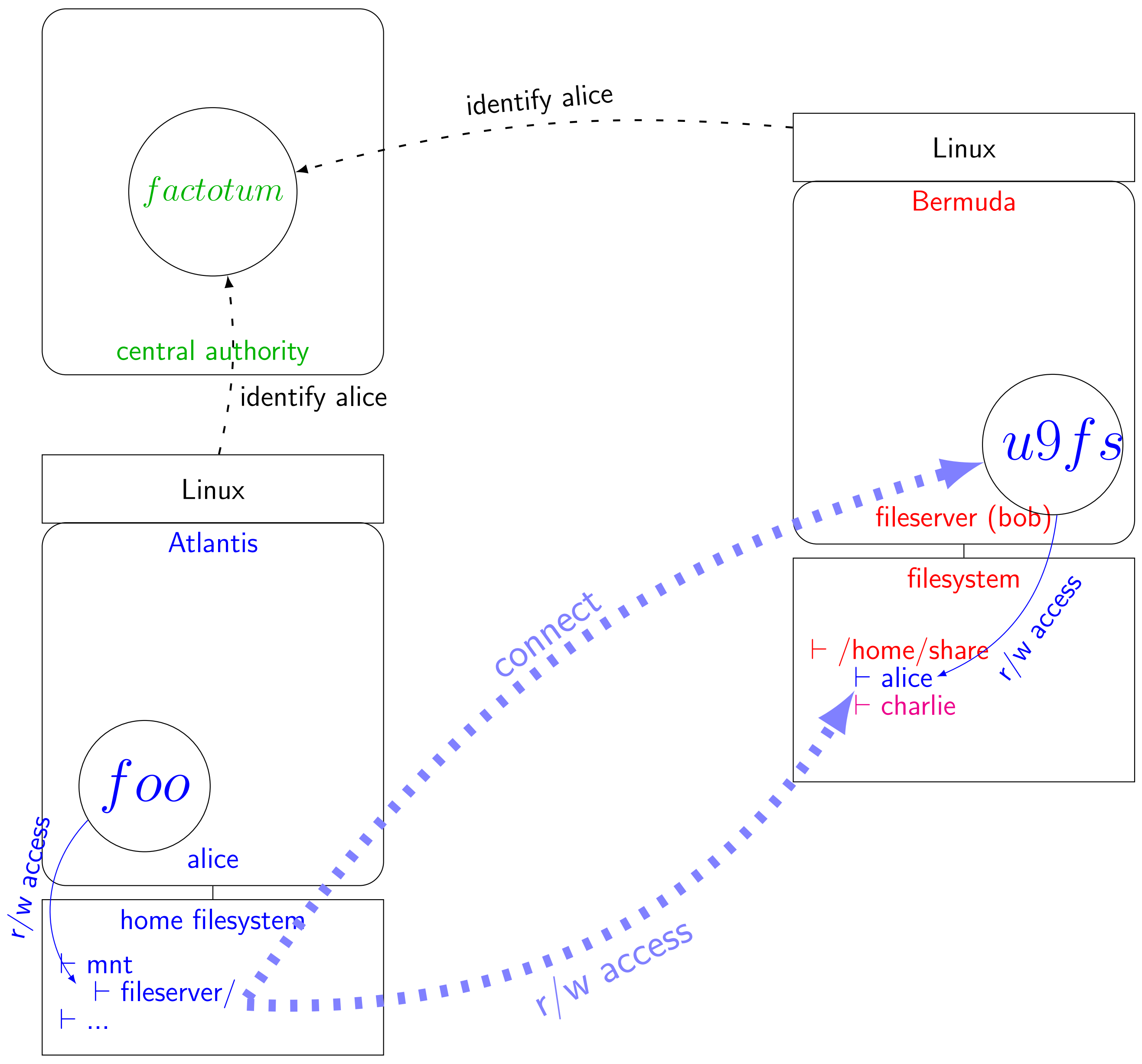


Figure 2: The `u9fs` process actually making changes in `Bermuda`'s file system on behalf of `foo` is owned by `alice`. This means that both kernel refer to the same central authority to agree on who `alice` is.

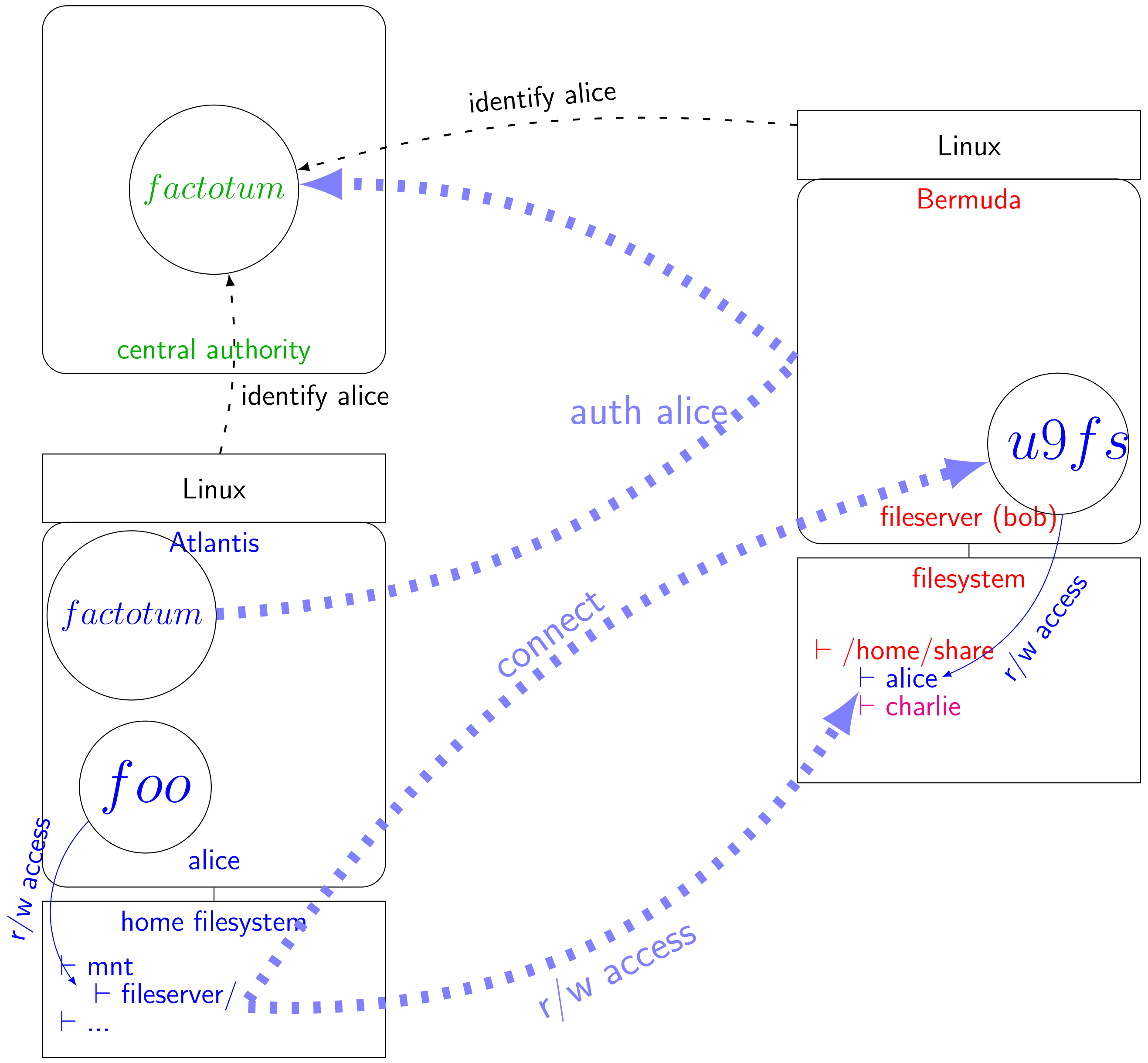


Figure 3: Authenticating `alice` on `Bermuda` is
a conversation between two `factotum` processes

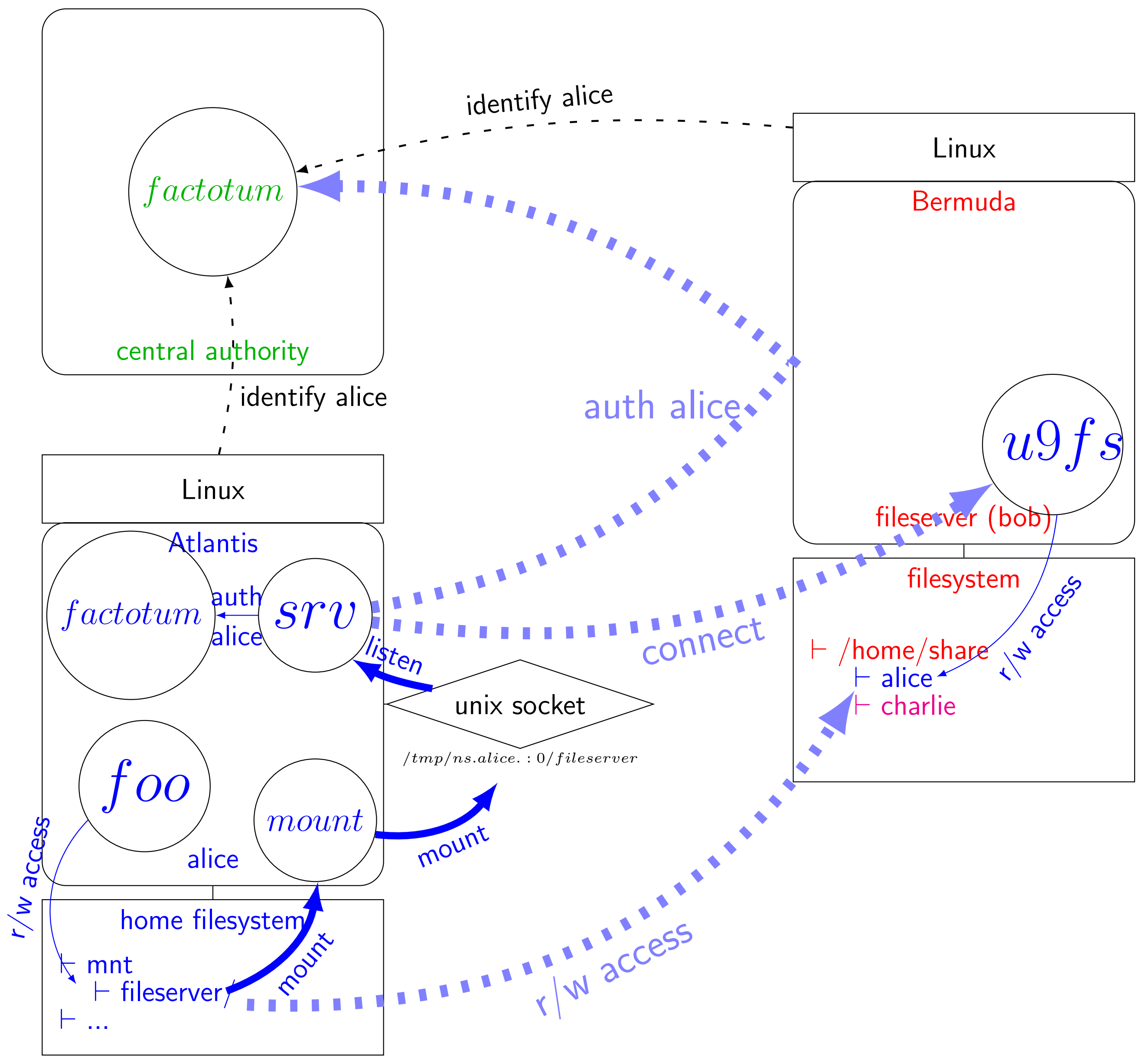


Figure 4: `srv` authenticates `alice` by relaying information between her `factotum` and the server, and then relays 9P messages between the server and the UNIX socket. `v9fs` talks 9P to the socket to offer a virtual directory on `Atlantis`' file system

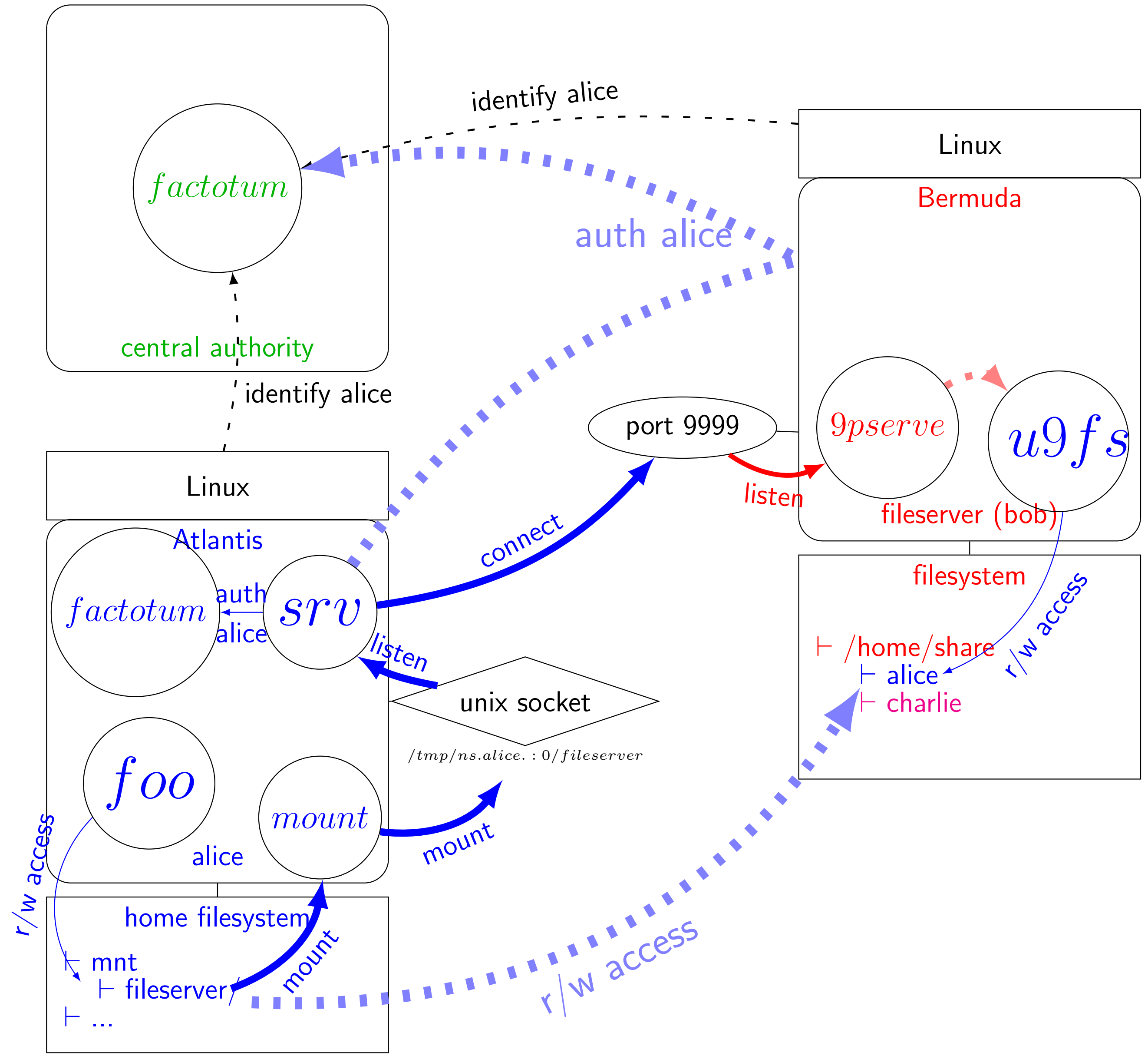


Figure 5: On `Bermuda`, `9pserve` multiplexes multiple clients (only alice is shown here) connections. Each client access their own server instance.

2.3. Security Agnosticism with vrs

2.3.1. What is vrs ?

On `Atlantis`, `srv` first authenticates as `alice` by relaying messages between her `factotum` and the server, and then simply relays 9P messages between the client process `foo` and the server.

One of our contributions is creating the `vrs` utility that mirrors this behavior on `Bermuda`: `vrs` first authenticates `alice` by relaying messages between `Bermuda`'s trusted `factotum` and the client, and then simply relays 9P messages between the client and the server process `u9fs`. See (Figure 6).

As Figure 6 shows, while the tools from `plan9port` were enough to ensure the *network transparency* of our setup, `vrs` is needed to ensure its *security agnosticism*. Whereas there are as many `srv` as there are clients, there is only one `vrs`. `vrs'` job is to authenticate each user and start a dedicated instance of `u9fs` owned by the corresponding user, then demultiplex the 9P flow coming from `9pserve` and forward the messages to the correct server instance.

2.3.2. vrs implementation

In Figure 7 and Figure 8, we show how three clients, Alice, Charlie and Dave, can send 9P messages to Bermuda and have their messages processed by each their own Network-Transparent, Security-Agnostic `u9fs` instance. We invite the reader to follow this explanation along with the circled letter spots (e.g. Ⓐ) on both figures.

Figure 7 shows three incoming messages:

- Alice's message is a Tauth, meaning she is not logged in yet. She is starting the authentication process.
- Charlie's message is a Tattach, meaning he has been successfully authenticated and is actually mounting the filesystem on his machine.
- Finally, Dave's message is a Twrite: After authenticating and attaching, Dave is actually using the filesystem.

9pserve These messages are partially rewritten by `9pserve`, Ⓐ, so that there is no collision between the values chosen by the various clients for their:

tag: When it receives an R-message, the tag lets the user know which T-message this R-message is a response to.

fid: A number that identifies a file. The server-side book keeping number is the qid.

afid The fid given in a Tauth becomes, after a successful authentication, a short-lived capability. It is called an afid. The server maintains a corresponding aqid.

`9pserve` maintains a mapping Ⓑ between incoming and multiplexed tags and (a)fids, in order to edit and demultiplex the returning R-messages to their correct clients Ⓝ.

Our new program `vrs` receives this stream of rewritten messages on its standard input Ⓒ, being oblivious to their respective emitters' location (*Network-transparency*).

Its job is to make sure these messages are legitimate, and demultiplex them by routing them to the correct subprocess.

Twrite The simplest example is Dave's Twrite message. `vrs` checks that the fid is mapped Ⓓ to an active and valid afid Ⓔ, which means that this Twrite comes after a successful Tauth and Tattach (and obviously one Topen, and probably some Twalk). It is therefore routed unmodified to Dave's subprocess Ⓕ, as it contains no cryptographic or authentication information.

Similarly, the Rwrite response from Dave's server Ⓘ is forwarded unmodified Ⓜ to Dave, save for `9pserve`'s demultiplexing Ⓝ, according to its mapping Ⓑ.

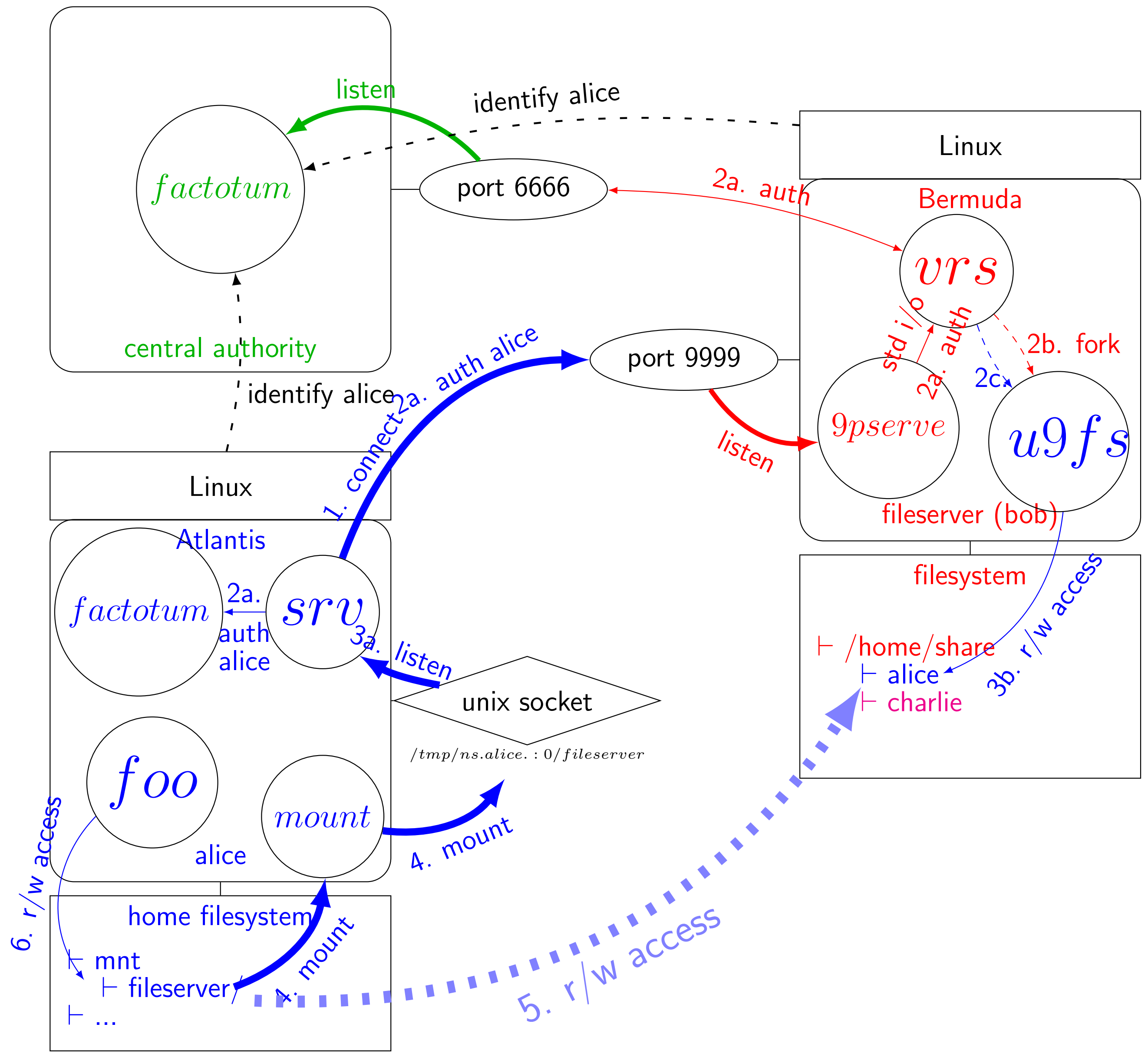


Figure 6: Our tool `vrs` is needed to ensure the security agnosticism of the server process (here, `alice`'s `u9fs` in `Bermuda`)

Tattach Upon receiving Charlie's Tattach message, `vrs` checks the given afid's PAM context Ⓔ (see subsection 3.1), and, if valid,

- forks Ⓖ,
- switches to uid 0, having temporarily dropped root privileges during normal operation,
- switches to `charlie`'s `uid` and `gid` as obtained through NSS (see subsection 3.3), permanently dropping root privileges in doing so,
- execs the 9P server, the main `vrs` process still keeping tabs on its standard input and output.

Once Charlie's server is up and running, `vrs`' fid mapping Ⓓ is updated to map `charlie`'s attach fid (here, 3) to its active and valid afid Ⓔ (here, 1) and to descriptors of `charlie`'s server's standard input and output.

The Tattach message is then stripped of its afid, which is replaced by the special value NOFID, which tells Ⓖ the newly started server process for `charlie` that no authentication is required. Thanks to `vrs`, Charlie's server is *security agnostic*, but being owned by `charlie`, it is protected by `Bermuda`'s kernel from outside interference and prevented from doing anything illegal.

It is of utmost importance to control the user-chosen afid's lifecycle. As Cox et al. (2002) put it:

> since [an afid] act like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

Its confidentiality during transport is ensured by TLS (see subsection 2.4), and the end of its lifecycle is dealt with by `vrs` intercepting any Tclunk message for an afid (as the server would not understand them anyway, being *security agnostic*) and unmapping the corresponding afid from Ⓔ. Any subsequent request referring to it either directly (Tattach) or indirectly (any other T-message with a fid mapped to this afid in Ⓓ) would fail.

Charlie's server's Rattach Ⓙ is transferred Ⓜ to Charlie, modified only for the demultiplexing Ⓝ, Ⓑ of its tag.

Tauth Alice's Tauth message makes `vrs` create a new PAM context in its mapping Ⓛ. The AQID returned in the Rauth Ⓚ message will let Alice's `factotum` run through its conversation with the `factotum` trusted by `Bermuda`: any subsequent Tread or Twrite messages sent by Alice on fid 0 will be multiplexed to fid 4 by `9pserve` and forwarded by `vrs` through our PAM module (see subsection 3.1) to `Bermuda`'s trusted `factotum`. `vrs` stays blissfully ignorant of the content of those read and write operations, which allowed us to edit `factotum` to add a new authentication protocol (see section 4) without modifying anything else.

Until the PAM context switched to SUCCESS, all Tattach on fid 4 will receive an Rerror from `vrs`.

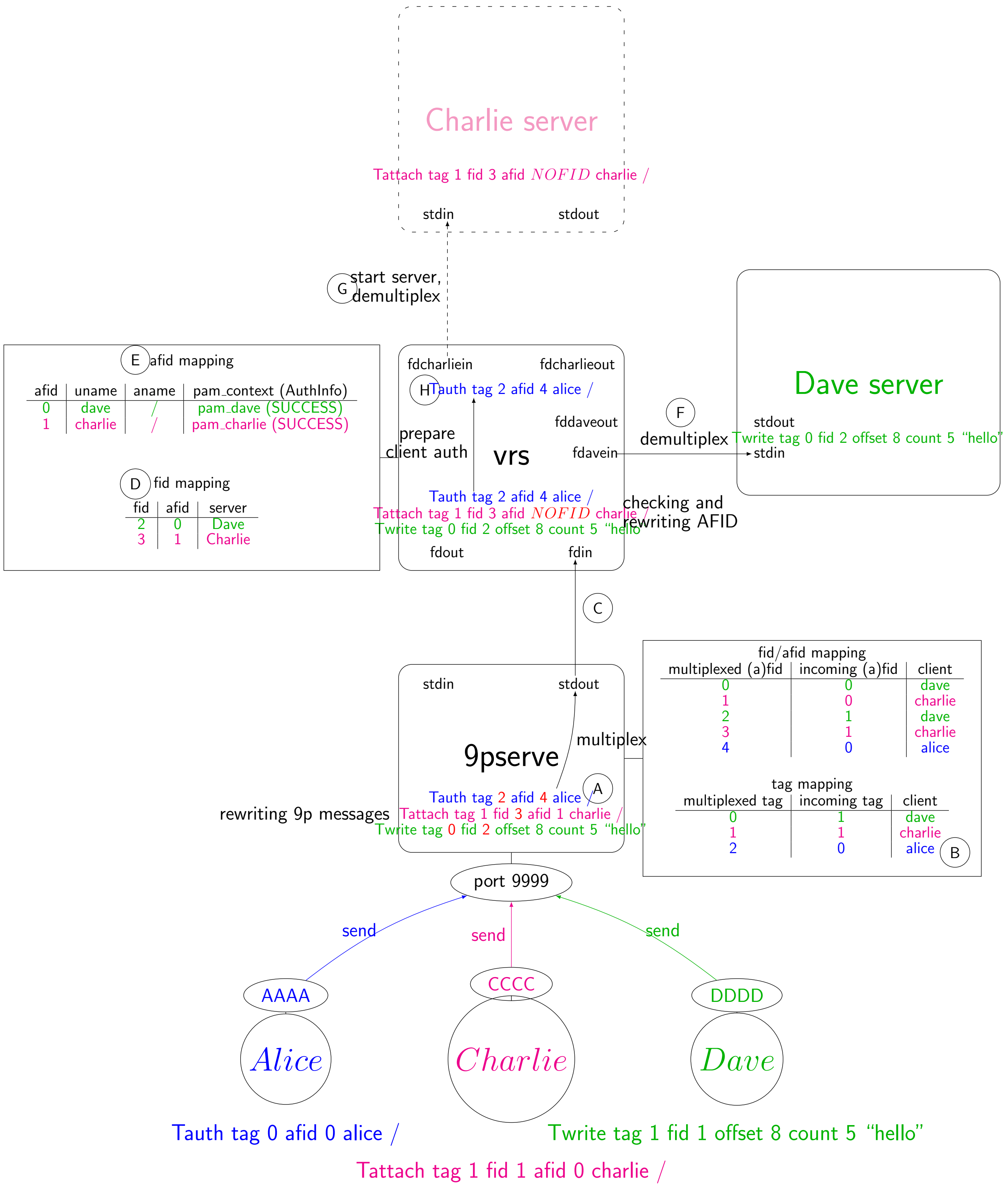
Until a successful Tattach happens, no server is even started for `alice`, thus reducing the attack surface.

2.4. TLS

Because afid are capabilities, 9P messages should not be transmitted in the clear. Yet, so far we refrained from talking about transport security. The reason is that this would have made Figure 1 to Figure 6 quite hard to read.

We use `stunnel` (Trojnara, 2019) to authenticate the server to the client and protect the confidentiality of the 9P messages against eavesdroppers.

On `Bermuda`, it is `stunnel` who is actually listening on port 9999. It decrypts the incoming messages and forward them to `9pserve`, who is listening on a UNIX named socket. That way no external attacker can connect to it, and the name of the socket is more descriptive than a port number.


Charlie server
Tattach tag 1 fid 3 afid $NOFID$ charlie /
stdin
stdout
G
start server, demultiplex
E
afid mapping
D
fid mapping
fdcharliein
fdcharlieout
H
Tauth tag 2 afid 4 alice /
fddaveout
prepare client auth
vrs
fdavein
Tauth tag 2 afid 4 alice /
Tattach tag 1 fid 3 afid $NOFID$ charlie /
Twrite tag 0 fid 2 offset 8 count 5 "hello"
fdout
fdin
F
demultiplex
Dave server
stdout
Twrite tag 0 fid 2 offset 8 count 5 "hello"
stdin
checking and rewriting AFID
C
fid/afid mapping
stdin
stdout
multiplex
9pserve
A
Tauth tag 2 afid 4 alice /
Tattach tag 1 fid 3 afid 1 charlie /
Twrite tag 0 fid 2 offset 8 count 5 "hello"
rewriting 9p messages
tag mapping
B
port 9999
send
send
send
AAAA
CCCC
DDDD
Alice
Charlie
Dave
Tauth tag 0 afid 0 alice /
Twrite tag 1 fid 1 offset 8 count 5 "hello"
Tattach tag 1 fid 1 afid 0 charlie /

afid mapping

afid	uname	aname	pam_context (AuthInfo)
0	dave	/	pam_dave (SUCCESS)
1	charlie	/	pam_charlie (SUCCESS)

fid mapping

fid	afid	server
2	0	Dave
3	1	Charlie

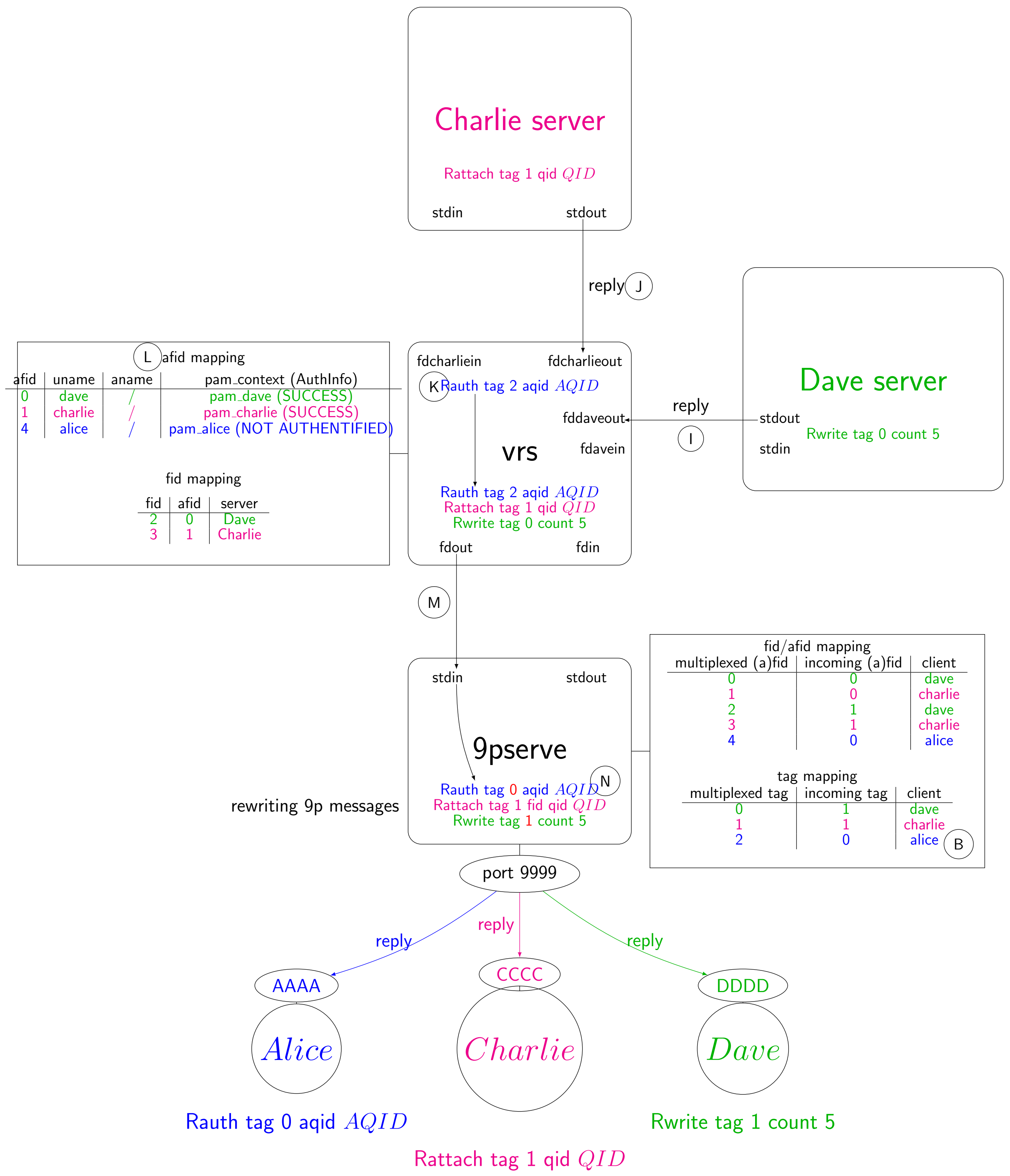
fid/afid mapping

multiplexed (a)fid	incoming (a)fid	client
0	0	dave
1	0	charlie
2	1	dave
3	1	charlie
4	0	alice

tag mapping

multiplexed tag	incoming tag	client
0	1	dave
1	1	charlie
2	0	alice

Figure 7: `9pserve` multiplexes 9P messages from multiple clients to a single conversation with `vrs`. `vrs` handles authentication and demultiplexes the client's messages to the corresponding *network transparent*, *security agnostic* server.



afid mapping

afid	uname	aname	pam_context (AuthInfo)
0	dave	/	pam_dave (SUCCESS)
1	charlie	/	pam_charlie (SUCCESS)
4	alice	/	pam_alice (NOT AUTHENTIFIED)

fid mapping

fid	afid	server
2	0	Dave
3	1	Charlie

fid/afid mapping

multiplexed (a)fid	incoming (a)fid	client
0	0	dave
1	0	charlie
2	1	dave
3	1	charlie
4	0	alice

tag mapping

multiplexed tag	incoming tag	client
0	1	dave
1	1	charlie
2	0	alice

Figure 8: R-messages are forwarded back to the clients. Authentication related messages are coming from our PAM module, other messages come from the client owned server.

On Atlantis, srv is not connecting to port 9999 of Bermuda, but to a named UNIX socket, served by another instance on stunnel, who is actually connecting to Bermuda.

We installed our own root certificate on all machines, so we can renew a server's certificate by simply signing it and installing it on the server machine, without having to tell all the clients about this change (as we had to when a server's host key changed when we used SSH).

3. System administration

Configuring a Linux system to trust a central factotum is as easy as establishing a secure connection to an authoritative server of one's choosing (we use stunnel for that, see subsection 2.4) and mounting factotum's virtual filesytem, *e.g.* in /etc/factotum/. We use the service managers of our systems to perform this task at startup.

3.1. PAM and factotum

vrs does not directly forward messages to the server's trusted factotum. Instead it calls our PAM module.

This makes implementing vrs a bit simpler, and is the expected way of adding an authentication mechanism on Linux. The main advantage is that it defers the choice of the authoritative factotum to the system wide configuration of PAM, where an administrator would expect to find it, and where it is protected from malicious interference.

This module will communicate with the authoritative factotum by reading and writing in the rpc file of the directory where factotum has been mounted.

The existence of this PAM module also let other application use factotum as an auth mechanism. We strongly discourage application code from dealing with security, but if there is no other way, a little bit of PAM boilerplate is probably the best way to go.

3.2. Factotum

Managing users is a simple matter of maintaining a configuration file of all users and their keys in a secure location (Plan 9 uses the secstore), and dumping it in the central authority's factotum's ctl file at startup.

Instead of storing secrets in the central authority, we elected to only store the users' public keys (see section 4), and have them store their private keys in a secure location, giving them to their own factotum instance when needed.

The ctl file understands and exposes records as lists of key-value pairs, *e.g.*:

```
key proto=p9sk1 dom=anonymous.lan user=alice...
```

Because vrs uses the p9any protocol, the client and the authoritative factotum will negotiate until a matching key is found between the client and the server.

Local administrators having access to the ctl file remotely is no big deal, as they only gain read access. Write access is only for the owner of the factotum process on the central authority server.

3.3. NSS and factotum

The last problem to solve is having all machines synchronize their users and groups database, which would normally be done by pointing to LDAP in the NSS configuration.

Instead, we modified factotum to make it serve two virtual files: passwd and groups with the same format as their namesakes in /etc.

These virtual files expose the same information as factotum's ctl file, in a different format.

We then just use the altfiles module to point NSS to where the authoritative factotum is mounted.

That way, user management (name, uids, groups, keys, etc.) is done all in one place: by writing the ctl file of the authoritative factotum (see subsection 3.2).

4. Adding the Guillou-Quisquater ZKIP to factotum

A lot of Plan 9's file servers use the `p9sk1` protocol, which relies on shared secrets held by a trusted third party, the authentication server. The authentication server uses a file server's secret to craft tickets relayed by the client's `factotum` to the server to prove the user's identity.

As a matter of policy, we don't like relying on sharing secrets. We elected instead to keep using one RSA key pair per user like we did when using SSH, and have the role of a central authority be endorsed directly by a `factotum` process remotely mounted by all server machines which chose to trust it (see section 3).

`factotum` already knows about RSA keys. We just added a new protocol that can use them, the Guillou-Quisquater zero knowledge proof (Guillou and Quisquater, 1988). By adding an RSA private key to its `factotum` with the `service=gq` attribute, a user can authenticate using this protocol to any service trusting a central authority `factotum` to which the corresponding public key has been added.

Once both `factotums` agrees on a key with the `service=gq` attribute, the client `factotum` sends a *commitment* computed from a random number and the public key to the server. The server answers with a random number of its own, which constitutes a *challenge*, and the client then sends a *response* computed from the initial random number, the private key and the *challenge*. The server then checks that this *response* is the same as another number computed from the *commitment*, the *challenge* and the *response*. If so, the client has proved its own knowledge of the private key (or is very, very lucky, but the process may be repeated) without disclosing information about it.

Thanks to the clean design of Plan 9's security model, adding this new protocol was just a matter of modifying `factotum`. The authentication transactions are sent back and forth unaltered between both `factotum` by `srv`, `9pserve`, and `vrs` (see Figure 6), which therefore need not be altered.

5. Related and future Works

5.1. Setuid

On Linux, a process asking a PAM module before switching its owner is just being courteous. There is nothing preventing a privileged process from switching its `uid` at will, which is why giving this power to application code (more likely to have bugs and security flaws) is a bad idea.

Privileged processes can now use Linux' capabilities system (same name, but not exactly the same thing as the capabilities we referred to when discussing 9P) to prevent themselves from doing certain classes of privileged actions. OpenBSD's `pledge()` has a similar aim and in our humble opinion a cleaner interface. In both cases, if a process can call `setuid()`, it can `setuid` to anyone.

Plan 9 is like France on the morning of August 5, 1789 in that privilege does not even exist anymore. Switching the file server's owner to `alice` is done by the host owner's `factotum` after she has earned its trust. `factotum` writes a capability to a file called `caphash` and gives it to the server process, which then writes it to a file called `capuse` making the kernel changes the server process' `uid` to `alice`. There is nothing else this capability is good for.

On Plan 9, some servers are not completely security agnostic because they must still use `auth`'s functions to switch their identity using this fine grained capability system, but

- the boilerplate is minimal (less than 10 lines, error management included),
- the server gains absolutely no other right than changing its owner to *e.g.* `alice` within the allotted time frame, whereas on Linux it could switch to anyone at anytime and do even more unless Linux' capability system is used to restrict its privilege somewhat (an uncommon case so far in most of the software we use),
- wrappers such as `rexexec` will switch identity before `execl`-ling to the truly security agnostic server.

In practice, some servers expecting UNIX clients or using deprecated forms of authentication such as `telnetd` or `ftpd` are not security agnostic at all. Others (e.g. `ctdfs`) are completely security agnostic and their actions will be limited by the kernel, given their owner's `uid`.

Plan 9's elegant capability-based mechanism has been ported to Linux (Ganti, 2008). Were this to be mainlined, we would use it on our machines and immediately thereafter modify `vrs` to use this system and therefore run unprivileged. We then would hunt all remaining privileged software and, if possible, convert them to use the `cap*` files as well.

5.2. Namespaces

On Plan 9, processes build their own composite view of the network-wide resources they need. This is done by mounting the needed services in one's so called *namespace*. This *namespace* is a per-process property. The `ns` utility can display the current *namespace* of a process as a script that can be run to recreate this *namespace* elsewhere. The commands are descriptive enough for the output of `ns` to also be used as documentation of the process' setup. Indeed it is almost always the name of a remote service that is used instead of the port number this service is listening on.

Our heavy use of SSH tunneling and sshfs led to port numbers or mount points conflicts. Port numbers made setup scripts hard to read and required careful synchronization between clients and servers.

Our switch to `vrs` made things a little bit better, as services can be named, making reading setup scripts easier. Also, `plan9port` provides the `namespace` utility, which can reasonably emulate the toe stepping avoiding functionality of Plan 9's per process namespace, but can not provide the secure isolation Plan 9 processes enjoy from one another.

Linux lacks this kind of process isolation. All processes share the same file tree, one has to be privileged to open a low numbered port, etc. This is being remediated by the slow but growing penetration of features like linux namespaces, cgroups and containerization. Albeit more awkward to use than the native namespaces in Plan 9, we think these features are exciting steps in the right direction.

Sadly, we know from successfully running most of the software our users want on Linux lxzones in SmartOS, which do not support complex namespace operations, that very few pieces of software make use of these functionalities, despite them having been introduced more than one decade ago.

Nevertheless, in the last couple of years, we achieved an almost Plan 9-like level of process isolation through the use of GNU Guix Courtès (2013), either as a package manager on top of our user's Linux distribution of choice, or as the whole system. In the process of creating a reproducible package management system, GNU Guix' creators have made first-class citizens of those seldom-used namespace, cgroups, and containerization features of Linux. By prefixing a command with `guix shell --container`, or simply using `call-with-container` in a service description file, one can enjoy on Linux the same level of process isolation as on Plan 9.

5.3. 9P vs. HTTP

Most of the software our users want expose their functionality through HTTP endpoints. Claiming RESTful properties (but often failing in subtle ways), when they implement authentication they do so by authenticating every transaction, awkwardly reimplementing the access control logic already available to them through the kernel.

We intended to work on a HTTP over 9P bridge, naturally mapping the HTTP endpoint to a file of the same name, with read calls translated as GET requests, write calls as PUT, POST or PATCH, etc.

Using this bridge, `alice` can authenticate, attach, walk to a endpoint of interest and open it with specific rights (read only, for example). The corresponding `fid` acts as a *de facto* capability and can be given away to another process for it to complete a very specific and narrow task, with no access whatsoever to anything else than what is strictly needed. `alice` can close the file and thus repudiate the capability as soon as she wants. This kind of capabilities is more fine grained and easy to use than any RESTful software access control we have seen in the wild.

On the backend side, if the software can have multiple instances of itself run concurrently on the same data files, we use `vrs` as described in this paper. The backend server's kernel deals with access control using the usual UNIX file permissions and the software is run *security agnosticly*.

This case is rare, however, and we more often than not end up having to run a single instance (typically database software like Postgresql or ElasticSearch do not support running multiple instances on the same data). Because our users are well behaved we run one instance per project, which gives them the ability to shoot one another in the foot. The alternative is to either somehow enforce access control at the bridge level, which increases the complexity of the bridge code and necessitates a way to tell the bridge about the permissions, or map with each software's own access control mechanism, which takes us back to square one.

Despite not getting any work done on this particular topic in the last couple of years, we still think this is a worthwhile avenue of research.

5.4. Sharing actual files

We tried NFS, AFS, SSHFS and Samba to share files. In terms of security and ease of use, the most serious contender to the solution exposed here was SSHFS, which we finally rejected because of `uid` management. NFS and AFS needs Kerberos (see subsection 5.6) to be secured, there is no AFS server for OpenBSD (which runs on one of our biggest fileserver) and Samba is very awkward to use without an Active Directory which we don't have.

All these solutions are battle tested and solve the file sharing problem day in day out everywhere across the world.

Comparatively, `srv` + `u9fs` may suffer from performance issues (although we have not run into that yet) and bugs, but are easier to install and expose a truly consistent access control policy to all users on the network. We are working on a BSD port which is currently at the experimental stage, not deployed on all our infrastructure.

Because sharing filesystems has become so easy, most of our workflows now rest on spools, sentinel files, named pipes, etc. Running those workflows locally or in a completely distributed way requires absolutely no change in the code, just a little bit of setup (mounting the remote resources) beforehand.

5.5. FUSE and 9P servers

The idea of exposing resources as virtual file systems is gaining traction on Linux thanks to FUSE. Applications such as sshfs or GlusterFS deal with data storage as "normal" files would, other such as encfs are overlays over data stored on the disk, and finally some other like perkeep (formerly camlistore), or Plan 9's plumber and acme expose an API, as modern software usually does nowadays with HTTP endpoints.

Plan 9 and Inferno are full of virtual filesystems to access things as diverse as *e.g.* emails, web pages or Lego Mindstorms. With Inferno especially, those can be started by `vrs` without having to port them to UNIX. One just starts, for each client, an embedded Inferno system running the service.

5.6. Kerberos, LDAP and SSH

We argue that our solution is easier to install, configure and administrate than the industry standards, Kerberos and LDAP. Furthermore, Kerberos is inherently and irremediably flawed in that it requires application code to handle security by sharing secrets with the Ticket Generating Service. This application code is likely not to be always up to date, or to have some flaw that would leak memory, thus granting an attacker access to a powerful secret.

By contrast, our solution handles security in a small number of well defined places that are suitably hardened (*e.g.* `factotum` will not let itself be debugged, nor will it swap its RAM (Cox et al., 2002)), while the servers and clients are built as completely *security agnostic*.

Furthermore, (a)fid are short lived and narrow capabilities, that will grant an attacker a lot less power than a Kerberos (Ticket-Generating) Ticket should he get ahold of them.

Finally, thanks to our implementation of the Guillou Quisquater protocol, our security may rest on asymmetric cryptography (our personal preference) instead of sharing secrets, once again limiting the attack surface.

6. Conclusion

Striving to bring the simplicity and elegance of Plan 9's security model to UNIX despite the latter's inherent limitations, we devised a distributed authentication setup where the only privileged process is our `vrs` daemon. This let us push all security and network code out of both client and server software, and places the burden of access control back on the kernel's shoulders, where it belongs.

Now, being entirely unprivileged, a buggy or compromised application process is extremely limited in the damage it can do. Furthermore, application code is now easier to write and configure, because it need care about neither networking nor security. This in turn reduce the number of bugs an attacker can exploit.

Through our use of the 9P protocol, application workflows run unmodified whether they are run locally or in a composite environment made up of resources from all over our network. Our modification of `factotum` as a central authority for the user and groups database ensures that the access control policy is consistent all over the network and that administration is dead simple.

This scheme has been deployed on our institution's network for weeks. Some applications still require SSH tunneling, but work is well underway on a HTTP over 9P wrapper, which will bridge the gap and let us make almost all resources we care about available as 9P servers.

Source code The code and the source of this paper can be found at `https://gitlab.com/edouardklein/vrs`

Thanks and Acknowledgements We would like to thank *Messieurs les Colonels* Duvinage et Nollet, for letting us enough leeway to work on our research despite pressing operational needs.

We owe a great deal to our friends and colleagues at the C3N, especially Jérôme Hénon, for their help in setting up the machines, and their willingness to try out strange software from the nineties (to be fair, a suprisingly high share of our hardware was from the nineties as well ;-).

References

Courtès, L. (2013). Functional package management with guix. `https://arxiv.org/abs/1305.4584`.

Cox, R. et al. (2003-2018). Plan9 from user space.

Cox, R., Gross, E., Pike, R., Presotto, D., and Quinlan, S. (2002). Security in plan 9. In Association, T. U., editor, *Proceedings of the 11th USENIX Security Symposium*, Available online: `https://9p.io/sys/doc/auth.html` (2018/04/04). Bell Labs and MIT LCS and Avaya Labs.

Ganti, A. (2008). Plan 9 authenitcation in linux. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*, 42:27–33. Available online: `http://gsoc.cat-v.org/people/ashwin/p9authLinux.pdf` (2018/04/04).

Guillou, L. C. and Quisquater, J.-J. (1988). A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In Guenther, C., editor, *Advances in Crytology*, pages 123–128. Available online: `https://cseweb.ucsd.edu/~mihir/papers/gq.pdf` (2018/07/07).

Jujjuri, V., Van Hensbergen, E., Liguori, A., and Pulavarty, B. (2010). Virtfs–a virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*, pages 109–120.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from bell labs. *Computing systems*, 8(2):221–254.

Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. (1992). The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–5. ACM.

Pike, R., Thompson, K., Presotto, D., Winterbottom, P., and al. (1987-2019). *Introduction to the Plan 9 File Protocol, 9P*. Alcatel-Lucent, Available online: `http://9p.io/magic/man2html/5/0intro` (2019/01/04).

Szeredi, M. e. a. (2010-2019). Fuse: Filesystem in userspace. `http://fuse.sourceforge.net`.

Trojnara, M. (2019). Stunnel. `https://www.stunnel.org/`.

Van Hensbergen, E. and Minnich, R. (2005). Grave robbers from outer space. using 9p2000 under linux. In The USENIX Association, editor, *USENIX Annual Technical Conference*, Available online: `https://www.usenix.org/legacy/events/usenix05/tech/freenix/full_papers/hensbergen/hensbergen.pdf` (2018/07/30).