

DD2497 Final Report - Group 5

Tommie Andersson, André Brogård, Henrik Kultala,
Hîvron Stenhav, Albin Winkelmann

January 12, 2022

1 Countered vulnerabilities

Microkernels often communicate, in some way, with external services for various reasons. By following the principle of least privilege (where each user, process etc has the least amount of privileges possible) you minimize the attack surface, and make it harder for potential malicious actors. Specifically for network traffic this can be accomplished through the usage of a so-called firewall.

The firewall is employed to ensure that only pre-defined traffic is allowed. This is a defensive mechanism to prevent remote exploits that could otherwise be possible on exposed processes, and prevent malicious processes from communicating with a remote host. Specifically it mitigates the effect of and prevents remote exploitation, where a remote adversary may try to gain access to the OS over the network.

With a firewall that can be configured on a per process basis already in place from previous years in this course, we have extended it to also allow firewall rules to only apply to specific users. By implementing firewall settings per user, this further improves the security of the microkernel as some users can have restricted internet traffic.

Firewalls that are implemented per-user are useful in many situations. For example, a large finance company could have employees who don't have enough knowledge about being safe online. Therefore, there could be root users in the company, in this case IT-technicians, that have more privilege than the other employees. These root users could have access to important websites that not everyone should have.

2 Design choices

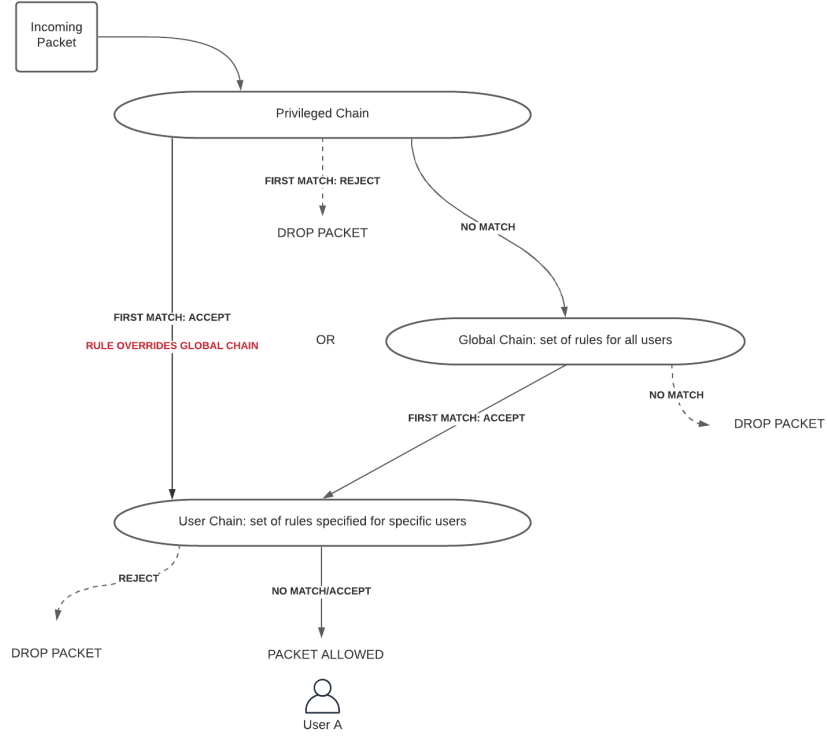
The Linux distribution comes packed with several firewalls, a popular one is *iptables*. *iptables* defines different chains, which are lists of rules to determine if an internet connection should be allowed, rejected or dropped. Policies will attempt to match packages coming in on properties such as IP (destination or source), port, protocol and more. The chains use a first-match approach, which means whichever rule first matches a packet will determine what to do with it. Therefore, the order of rules matter: default, or catch-all, rules are of course placed at the end. The first-match approach has significant performance advantages for complex firewalls that have a lot of rules.

The previous firewall that we built our implementation upon used a different approach, where each rule had an associated priority level. This allowed all new rules to simply be added to the beginning of the rules list, and then prioritizing between multiple matching rules were done through the priority field (with same priority going with a last-match approach). Furthermore, all rules were placed in either the *in_rules* list or the *out_rules* list, based on if the rule regarded incoming or outgoing traffic.

However, this has the drawback of having to check every single rule (of either the “in” or “out” direction) every time, since we can't know if there is a matching rule with a higher priority later in the list. Because of this, we chose to modify the existing firewall to function more like *iptables*, with multiple chains that all utilize a first-match approach.

The general idea is that we have different “levels” of chains, which are considered in order. That is, we first consider the first chain, and when we find a matching rule/have traversed the entire chain without finding a matching rule, we continue to the second chain based on the outcome. In total we have 3 chains.

Figure 1: Design Choices of the chains



The first chain is a set of rules that privileged users can edit to override non-user specific rules. These rules can be either ACCEPT or REJECT policy and when going through the chain, we use the first-match technique, so as soon as there is a matching rule we follow its policy, even if there are other matching rules further down the chain. The main purpose of this chain is to for instance allow admins to set up more lenient rules for themselves.

The second chain is the global chain, which every user is affected by. This chain functions the same way as the previous privileged chain, except that in the global chain all rules apply to all users of the system. The global chain is only considered if we don't find a matching rule in the privileged chain. The intention with this chain is to allow a set of standard rules for a typical user. If no matching rule is found in any of these two chains, we go with a whitelist approach and the packet is dropped.

Essentially, these first two chains function like one entity, with the first matching rule over any of the two being chosen, and if no matching rule is found after traversing both the packet is dropped. The separation into two chains is for clarity and to ensure that if you add a rule meant to override the global chain, it is placed in the privileged chain (which is considered first).

The third, and last chain, is where we have user specific rules for regular users. These rules will only be considered if a packet matched a rule with the ACCEPT policy in one of the previous two chains. This allows non-privileged users to further restrict their traffic if they consider the global rules too lenient. Since only packets allowed by the first two chains reach this chain it cannot be used to increase the lenience of a user's traffic. Because packets that reach this chain are already allowed either on a system level (the global chain) or on a level overriding the system rules (the privileged chain) we consider it safe to use a blacklist approach on this chain, letting through packets that don't match any rule in this third chain. This means that users that are not interested in customizing their own firewall rules can just leave this chain empty without any issues. This is part of the intention with this chain: to allow regular, non-privileged users to block additional traffic if they think the standard, global rules are too lenient. But it shouldn't be required of a regular user to fiddle with this chain (or indeed the firewall at all) if they don't want to.

Since editing the first two chains allows a user to increase network access of any user, they can only be edited by privileged users. Meanwhile, as mentioned in the last paragraph the third chain can't be used to gain more access so it can safely be edited by any user.

The conversion to using chains with a first-match approach and the addition of a user ID to the rules required us to change several signatures of firewall functions and to update the low level implementation of handling firewall rules. For instance, we changed the add and remove functions to utilize an index of where to add/remove a rule. But other than that, we tried to reuse the old implementation and build upon it, rather than replacing it. The firewall is still a server that intercepts network packets and inspects them, and it still has additional checks for incoming TCP packets (we haven't touched this check), and we kept the Command Line Interface (CLI) tool for the firewall (though this required some updates to work with our new structure). This CLI tool allow users to dynamically add, remove and list rules of the firewall without having to restart the entire firewall server.

3 Implementation details

Our first step in adding user ID:s to the firewall rules was to figure out how to gain the correct user ID for a given network packet. In Linux system, processes need to know who is executing a particular process. In Minix, the Process Manager (PM) holds this information. Because users can execute on behalf of other users, and change user, one variable is not enough: three variables are used. The real UID is always the one who is executing, regardless if this is on behalf of some other user. If we are user A and if we execute on behalf of another user B, their user ID will be the effective UID, while the real ID would be our user ID: user A. The saved variable is used internally to handle cases where you need to change user and you do not want to lose information of who it is. It is standard that all programs will verify that the effective UID has correct permissions, and of course our firewall is no different.

Thus, we needed a way to get the effective user ID of a network packet. By looking at the previous firewall implementation we saw that for a given network packet they utilized endpoint information and made a SYSCALL to the PM to retrieve the associated process name. Since the user ID info is also handled by the PM, we added a very similar SYSCALL to the PM that instead retrieved the effective user ID, given endpoint information. This required incrementing the number of SYSCALL:s for the Pm in *minix/include/minix/callnr.h* and *minix/servers/pm/table.c*, adding a new IPC message type in *minix/include/minix/ipc.h*, adding a wrapper function for the SYSCALL in *minix/lib/libsys/getepinfo.c* (and its declaration in *minix/include/minix/syslib.h*), and implementing the SYSCALL in *minix/servers/pm/misc.c* (and declaring it in *minix/servers/pm/proto.h*).

After having access to the user ID we started working on our chains. Like the previous implementation we kept using linked lists stored in the heap for our firewall rules, but we changed the structure. Previously there were two lists: the first consisting of rules for incoming packets and the second consisting of rules for outgoing packets. We saw no need to keep these lists separate, especially knowing that we would already have multiple lists representing our different chains, so we removed this distinction. Then we changed the abstraction a little: the previous implementation was saving the list by knowing the head entry and having every rule contain a pointer to the next and previous rules; making it a doubly linked list. Instead we made a struct representing a chain that wraps a chain ID and a head entry, and we made all entries only have pointers to the following and previous entries, as well as having a pointer to the actual rule. That is, we separated the logical entities of “chain”, “entry”, and “rule” into different physical entities through different structs, making the separation between them clearer. We also kept the property of the list being doubly linked rather than singly linked, although admittedly we currently do not make use of this.

A rule in the firewall is similar to the previous implementation, but instead of having a priority field (and having pointers to the next and previous rules) it has a user field for the user ID and a direction field (since we no longer have different lists for rules regarding incoming or outgoing packets). In total, this is what a current rule looks like:

```
fw_chain_rule {
    uint8_t type;
    uint8_t action;
    uint8_t direction;
    uid_t user;
    uint32_t ip_start;
    uint32_t ip_end;
    uint16_t port;
    char p_name[MAX_NAME_LEN];
```

, where *MAX_NAME_LEN* is the longest string allowed for a process name (defined to be 16 bytes by the previous project).

When it comes to operating on rules (adding and removing) our implementation is somewhat different from the previous one. Previously rules were always added at the start of the list, and removing a rule meant looking for the first rule whose fields matched all the arguments to the remove function. We operate on an index basis instead, inserting a rule at the given index, and removing a rule at a given index. This means that removing a rule only takes a chain ID and an index, rather than specifics for all fields. When adding a rule, if the given index is invalid (outside of the list size) then the rule is simply added at the end of the list. For instance one can give the index -1 to always add a rule at the end. If the index is invalid when removing a rule, the list is left untouched and the function only states this in a print and returns.

For matching a rule, the different chains are probed in the order described in the “Design choices” section. For each rule in a chain, it is checked if all fields (except the action field) match the arguments, or if a field of the rule has the “any-value”. As the name implies such a field will match any value. If all fields match, then that rule is chosen and its action field determines what happens next (either ACCEPT or REJECT).

The “any-values” are generally defined to be 0 or the null-string (with the exception of user ID since a user can have the ID 0). Thus, when defining enum-like constants (like *IN_RULE* and *OUT_RULE*) we tried to avoid using 0 since it might clash with our “any-values”. Thus, the enum-like constants we have used start at 1 instead of 0. Speaking of which, we have also tried to define constants for most places where we otherwise would have used “magic constants”. This to increase readability and to reduce errors when making changes.

We have generally tried to store all our defined constants of the “fwdec-server” as well as the definition of the chain-related structs in the file *minix/servers/fwdec/fwchain.h*, to keep them in one place. Some of these were also defined in other places relating to the command line utility, or the interface functions other servers would use to contact the “fwdec-server”. Because of this, some of the needed constants were removed from the *fwchain.h* file and were instead imported from the *minix/include/sys/fwctl.h* header file.

When it comes to the Command Line Interface (CLI) we could keep most of the previous logic, but we still had to update several files since we changed how rules are added and removed, and what fields they contain. In *minix/commands/firewall/firewall.c* we updated the actual command line command, in *minix/lib/libc/sys/fwctl.c* (and *minix/include/sys/fwctl.h*) we updated the wrapper functions for making a SYSCALL to the Virtual File System (VFS) (which forwards the IPC-message from the command line utility to the “fwdec-server”), in *minix/include/minix/ipc.h* we updated the IPC-message carrying info about an issued command line command, in *minix/servers/vfs/misc.c* we updated how the VFS forwards the IPC-message from the command line utility to the “fwdec-server”, and lastly in *minix/servers/fwdec/main.c* and *minix/servers/fwdec/fwdec.c* (and in *minix/servers/fwdec/proto.h*) we updated how the “fwdec-server” handled received IPC-messages from the command line utility.

Lastly it is worth mentioning the issues we had with our IPC-message for the CLI. We carefully made sure that all fields would add up to 56 bytes, and yet it mysteriously complained when we did this, and sometimes it compiled when it didn’t add up to 56 bytes. In the end we managed to find some consistency indicating that every field of the message-struct had to be contained within one aligned 32 bit word; meaning if our first field was only 1 byte and the second field was 4 bytes, there would be a gap of 3 bytes in our message since the second field would not want to be split over the 3 remaining bytes of the first word and then have its last byte in the next word. Similarly for the entire message, it was allowed to be up to 3 bytes too small, presumably since the next item after it would be aligned to the next free word.

While this is a little difficult to properly explain, and the actual cause is still just a theory on our part, the important takeaway is that the order of the fields in the IPC-message matter. By ensuring that all our fields never ended up “between words” it worked as expected, allowing us to pad the message with the expected amount of padding.

4 How our firewall counters target vulnerabilities

So let's summarize how our firewall implementation achieves its goal. Like the previous implementation, our firewall still intercepts all network packets (both incoming and outgoing) and forwards them to the firewall decision server ("fwdec"). New is that the user ID is also included in the info sent to the decision server. This decision server iterates over all firewall rules contained in the different chains. These different chains allow adding rules that affect different users in different ways: global chain for all users, privileged chain to override this, and user chain to further restrict one's own traffic without needing privilege, as described in section "Design choices". Thus the firewall can both contain rules that affect all users, and rules that only affect specific users. When the decision server finds a matching rule it acts according to the action specified by that rule, or, if no matching rule is found, it acts according to the specification outlined in the "Design choices" section.

To prevent privilege escalation unprivileged users can't modify anything but the rules affecting themselves in the user chain (only privileged users can modify the first two chains); which can only further restrict the access they already have. One important aspect in achieving this is that if no matching rule is found in the first two chains we simply drop the packet without ever going to the user chain, meaning packets that aren't explicitly allowed won't be tested against the chain that unprivileged users can modify.

Thus, our firewall inspects all network packets and can treat them differently depending on what user is executing the process they are associated with. And privileged users can modify the firewall dynamically, while unprivileged users can only modify it in such a way that they further restrict their own access.

5 Future improvements

If the reader is looking at the source code of the project, note that we have inserted TODO:s called "TODO5" (TODO for group 5) to allow us to search for them in the project (there were several previous TODO:s meaning only searching for "TODO" yielded many to us uninteresting results). While we solved the most important ones, some are left as remarks about things we would have liked to do something about but didn't have enough time to. If looking to continue and/or improve our work this is a good place to start.

Otherwise, the main improvement that we thought of but did not have time to implement was the addition of specifying firewall rules for specific groups of users. This would present the issue of having to somehow prioritize firewall rules for the user and for all different groups the user is part of. Should one take the first rule found from either the user or any of the user's groups, or should there be some different prioritization? The group issue could also regard access control of the firewall: could a user be considered unprivileged but be in a privileged group that should be able to edit the firewall?

6 Contribution

In our project we have pair-programmed a lot, meaning we generally were multiple people contributing to the same thing at the same time. Thus, my contribution outlined below will overlap with the contributions of other group members.

6.1 Initial user-aware firewall

In the first part of the project we mostly worked together as the entire group. We inspected the code of the previous project, discussed what we would keep and what we would change, and how we should implement our extensions. Then we started implementing our version and tried to figure out how to get it to work in Minix.

During this part I, like the rest of the group, partook in the discussions forming our design and wrote code for the implementation of our chains of rules. This involved understanding the previous implementation of adding, removing, listing, and matching rules, and modifying it to fit our needs. We also added a new SYSCALL to the PM for getting the user ID of an endpoint, by looking at a SYSCALL with similar purpose that the previous group implemented.

6.2 Structural improvements and command line interface

After the seminar presentation we split up into smaller groups, meaning we generally worked in groups of 2 or 3 people. My group mates and I focused mainly on 2 areas: improving the structure of the code and updating the command line interface to work with our new structure.

6.2.1 Structural improvements and fixes

The most important part of this task was that we added an ID field to the chain structs, allowing us to define multiple chains that could be handled differently. This part of the task was quickly done. However, we had noticed that the “fwdec-server” was not necessarily clear and consistent in how it was written. This had hardly gotten better when we tried to make our own additions (now and in the initial phase), without properly knowing what decided where different entities were placed in the different files. Thus, we set out to improve the “fwdec-server” so that it would be easier to find related entities, easier to understand and less error prone to modify.

This included:

- Making the *fwrule.h* file into a proper header file by extracting all implementation to a new *fwrule.c* file.
- Removing several unnecessary include-statements in multiple files.
- Adding some necessary include-statements to some files to make them less brittle: previously some header-files did not include everything they needed but they themselves were always included after their requirements in other files meaning the code worked, as long as you didn't reorder the include-statements. However this brittleness became painfully apparent when formatting the files, which among other things sorted the include-statements in alphabetical order and, thus, breaking the delicate order needed.
- Moving most constants and definitions of the “fwdec-server” (we left some that we never touched in our implementation) to a separate file so that one could easily find all of them in one place.
- Refactoring enum-like constants so that they start at 1 instead of 0, since we used 0 to mean “any value” in multiple places.
- Defining more constants so that we rely less on magic constant in the code. For instance we now have all the “any value” constants defined (which generally are 0 or the null string, with the exception of user ID).
- Adding documentation to most functions we implemented or updated (from now and the initial phase), as well as commenting the implementation at times to make it easier to follow.

Furthermore we also found bugs in our own implementation of some low level “chain_rule” functions (*add_chain_rule* and *remove_chain_rule*) that had to be amended.

6.2.2 Command Line Interface

This work was done in parallel with the structural improvements and fixes, by the same people. In getting the Command Line Interface (CLI) to work with our new structure we had to make changes in several files. While it generally wasn't too much code to write, sometimes it took quite some time to figure out how to make it work. For instance, the firewall command line utility is only sparingly documented so fully understanding how it worked and modifying it took some tinkering.

But most notably we had quite some issues modifying the IPC-message that was utilized by the CLI. As described in the “Implementation details” section, what constituted a valid message was quite curious, and we spent a lot of time trying to figure it out.