Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

The RAppArmor package: Linux Based Sandboxing of an R Process in a Shared Environment

Jeroen Ooms
UCLA Dept. of Statistics

Second Author
Plus Affiliation

Abstract

With the increasing availability of public cloud computing facilities and scientific super computers, there is a great potential for making R available through public or shared resources. This allows researchers to efficiently run code requiring a lot of cycles and memory, or embed R functionality into e.g. systems or web services. However some important security concerns needs to be addressed before this can be put in production. The prime use case in the design of R has always been single statistician running R on the local machine through the interactive console. As a result there are practically no restrictions on what the user is allowed to do with the operating system, which could potentially result in malicious behavior or excessive use of hardware resources in a shared environment. Properly securing an R process turns out to be a complex problem. We describe some approaches of securing R and illustrate potential issues using some of our personal experiences in hosting public R services. Finally we introduce the **RAppArmor** package which provides a Linux based reference implementation for sandboxing R on the level of the operating system.

Keywords: R, Security, Linux, Sandbox, AppArmor.

1. Security in R: Introduction and Motivation

The R project for statistical computing and graphics (R Development Core Team 2012) is currently one of the primary tool-kits for scientific computing. The software is widely used for research and data analysis in both academia and industry, and is the de-facto standard among statisticians for the development of new statistical computation methods. With support for all major operating systems, a powerful and stable codebase, more than 3000 addon packages and a huge active community, it is fair to say that the project has matured to a production-ready computation tool. However, one thing that is somewhat surprising is that the way in which R is used, has hardly changed since its initial design. Even though internet access, public cloud

computing (Armbrust *et al.* 2010), live and open data and scientific super computers have transformed the landscape of data analysis, R is still almost exclusively used as an end-user tool, running on the local machine of the researcher, operated through the interactive console. This seems somewhat of a missed opportunity. The demand for data analysis tools has never been higher, and many open source systems and software stacks could benefit greatly from the high quality analytical capabilities that R has to offer.

One reason developers are reluctant to build on R are concerns regarding security and management of shared hardware resources. Reliable software systems require components which behave predictably and cannot be abused. Because R was primarily designed with in mind a single user running the software on the local machine through the interactive console, security issues and unpredictable behavior have not been considered a major concern in the design of R itself. Hence, these problems will need to be addressed somehow before software developers will feel comfortable making R part of their infrastructure, or convince administrators to use their facilities to expose R based services to the public. It is our personal experience that the complexity of these issues is easily underestimated when designing stacks or systems that build on R. There are a number of issues that are very domain specific to scientific computing, which makes building on R quite different from embedding other software or languages. This might explain the limited adoption of R as a computational back-end engine so far.

1.1. Security when using Contributed Code

Building systems on R has been the main motivation for this research. However, security is becoming a concern for R in other contexts as well. As the R community is growing rapidly, it becomes more unsafe to rely on the social aspect of contributed code. For example, on a daily basis, dozens of packages and package updates submitted to CRAN. These packages can contain code written in R, C, Fortran, C++, Java, etc. It is practically impossible for the CRAN maintainers to do a thorough audit of the full code that is submitted, every time. Some packages even contain pre-compiled Java code for which the source is not included at all. Furthermore, R packages are not signed with a private key as is the case for e.g. packages in most Linux distributions, which makes it hard to verify the identity of the author. As CRAN packages are automatically build and installed on hundreds, maybe thousands of machines around the world, they become a more interesting target for people with questionable motives. Hence there is a real risk of packages containing malicious code making their way unnoticed into CRAN. Risks are even greater for packages that are distributed through channels without any form of code review, for example via email or through the increasingly popular Github repositories (Torvalds and Hamano 2006; Dabbish, Stuart, Tsay, and Herbsleb 2012).

In summary, it is not unreasonable for the R user to be a bit cautious when installing and running contributed code downloaded from the internet. However, things don't have to be as serious as described above. Thinking about security is a good practice, even if there is no immediate cause for concern. Some users simply might want to protect against themselves, making sure they don't erase any files by accident and that R does not interfere with other activities on the same machine. Knowing that R is running with no unnecessary privileges can be reassuring to the user and system administrators, and might one day prevent a lot of trouble.

1.2. Sandboxing the R environment

This paper explores some of the potential problems, along with approaches and methods of securing R. Some of the different aspects and concerns of security in the context of R are illustrated using personal experiences, or examples of bad or malicious code. We will explain how untrusted code can be run inside a *sandboxed* R process. Sandboxing in this context is a somewhat informal term for creating an environment in which untrusted software runs without capabilities of doing anything harmful to the system. As it turns out, R itself is not very suitable for managing access control policies, and the only way to enforce security properly is by leveraging features from the operating system. To exemplify this approach, a reference implementation based on Linux and AppArmor is provided which can be used on Linux distributions as the basis for a sandboxing toolkit. This package is used throughout the paper to show some of the issues could be addressed.

However, we want to emphasize that we don't claim to have solved the problem. This paper mostly serves an introduction to security for the R community, and hopefully creates some awareness that this is a real issue moving forward. The **RAppArmor** package is one approach and a good starting point for experimenting with dynamic sandboxing in R. However it mostly serves as a proof of concept of the general idea confining and controlling R in order to extend the applicability. The paper describes some examples of use cases, issues, policies and personal experiences which give the reader a sense of what is involved with this topic. Without any doubt, there are other concerns beyond the ones mentioned in this paper, many of which might be specific to certain applications or systems. We hope to invoke a discussion in the community about potential security threads and solutions related to using R in different scenarios, and encourage those comfortable with other platforms or use R in different contexts to join the discussion and share their concerns, experiences and solutions.

2. Use-Cases and Concerns of Sandboxing R

Let us start by taking a step back and put this research in perspective by describing some example use cases where security in R could be a concern. Below 3 simple examples of situations in which it is useful to be able to run R inside a sandbox. The uses cases are increasing in complexity, and require more advanced sandboxing methods.

Running Untrusted Code

Suppose we found an R package in our email or on the internet that looks interesting, but we are not quite sure who the author is, and if the package does not contain any malicious code. The package is too large for us to inspect all of the code manually, and furthermore it contains a library in a foreign language (e.g. C++, Fortran) for which we lack knowledge to really assess what exactly is going on. We would like to give the package a try, but without exposing ourselves to the risk of potentially jeopardizing the machine.

One solution would be to run the code on a separate or virtual machine. However this is somewhat cumbersome and we will not have our regular workflow available. In practice creating new machines is a bit unpractical and not something that we might want to do on a daily basis. It would be easier if we could just sandbox our regular R environment for the duration of installing and using the new package. If the sandbox flexible and unobtrusive enough to not interrupt our daily workflow, we could even make a habit out of using it every

time we use contributed code (which for most users is every day).

Shared Resources

Another use case could be a scenario where multiple users are sharing a single machine. For example, a system administrator at a university is managing a big computing resource and would like to make it available to faculty and students for using R. This way they could run R code that requires more computing power than their local machine can handle. For example a researcher might want to do a simulation study, and fit a complex model a million times on generated datasets of varying properties. On her own machine this would take months to complete, but the super computer can finish the job overnight. The administrator would like to give this and other researchers an automated way to run their R code on the supercomputer. However he is concerned about users interfering with each others work, or breaking anything on the machine. Furthermore he wants to make sure that system resources are allocated in a fair way so that no single user can consume all memory or cpu on the system.

Embedded Systems and Services

There have been a number of efforts to facilitate integration of R functionality into various 3rd party systems. Some examples of interfaces from popular general purpose languages are `RInside` (Eddelbuettel and Francois 2011), which embeds R into C++ environments, and JRI/REngine `JRI` which embeds R in Java software. Similarly, rpy (Moreira and Warnes 2006) provides a Python interface to R, and RinRuby is a Ruby library that integrates the R interpreter in Ruby (Dahl and Crawford 2008). Littler provides hash-bang (i.e. script starting with `#!/some/path`) capability for `R` (Horner and Eddelbuettel 2011). The Apache2 module RApache (`mod_R`) (Horner 2011) makes it possible to run `R` scripts from within the Apache2 web server. Heiberger and Neuwirth (2009) provide a series of tools to call R from DCOM clients on Windows environments, mostly to support calling R from Microsoft Excel. Finally, `RServe` is TCP/IP server which provides low level access to an R session over a socket (Urbanek 2011).

The third use case originates from these developments: it can be summarized as confining and managing R processes inside of embedded systems and services. This use case is largely derived from our personal experience: we are using R inside a number of systems and web services that provide on-demand calculations and plotting over the internet. These services have to respond quickly and with minimal overhead to incoming requests, and should scale to serve many jobs per second. Furthermore the systems have to be stable, requiring that jobs should always return within a given timeframe. Depending on user and the type of job, different security restrictions might be appropriate. Also we need a way to dynamically enforce limits on the use of memory, processors and disk space on a per process basis. These requirements demands a more flexible and finer degree of control over the process privileges and restrictions than the first two use cases. It encouraged us to explore more advanced methods than the conventional tools and forms the most central motivation of this research.

2.1. System Privileges and Hardware Resources

The use cases described above provide motivations and requirements for an R sandbox. Two inter-related problems can be distinguished. The first one is preventing system abuse, i.e. use of the machine for malicious or undesired activities, or completely compromising the machine.

The second problem is managing sharing of hardware resources, i.e. preventing excessive use of resources by limiting the amount of memory, cpu, etc that a single user or process is allowed to consume.

System Abuse

The R console gives the user direct access to the operating system and does not implement any privileges restrictions or access control to prevent malicious use. In fact, some of the basic functionality in R actually assumes quite profound access to the system, e.g. read access to system files, or the privilege of running system shell commands. Therefore, running user supplied R code without any restrictions can get us in serious trouble. For example, the code could call the `system()` function which provides an interface to the system shell. From here any system commands can be executed, which can potentially be harmful. But also innocent looking functions like `read.table` can be used to extract sensitive information from the system, e.g. `read.table("/etc/passwd")` will gives us a list of users on the system or `readLines("/var/log/syslog")` shows system log information.

Even an R process running as a non-privileged user can do a lot of harm. Some potential issues are code that contains or downloads a virus or security exploit, or searches the system for the users personal information. Appendix B.2 shows an example of a simple function that searches the users home directory for documents containing credit card numbers. Another increasing global problem are viruses that make the machine part of a so called botnet (Abu Rajab, Zarfoss, Monrose, and Terzis 2006). Once infected, the compromised machines ("bots") connect to a centralized server and wait for instructions from the owner of the botnet. Botnets are mostly used to send spam or to participate in DDOS attacks: centrally coordinated operations in which a large number of machines on the internet is used to flood a server or provider with network traffic with the goal of taking it down by overloading it (Mirkovic and Reiher 2004). Botnet software is often invisible to the user of an infected machine and can run with very little privileges: just network access is sufficient to do most of the work.

When using R on the local machine and only running our own code, or from trusted sources, these scenarios might sound a bit far fetched. However, when running code downloaded from the internet or exposing systems to the public, this is becoming a real concern. Internet security is a global problem, and there are a large number of individuals, organizations and even governments actively employing increasingly advanced and creative ways of gaining access to others machines. Especially servers that run on beefy hardware or fast connections are attractive targets for individuals that could use these resources for other purposes. But also servers and users inside large companies, universities or government institutions are frequently targeted with the goal of gathering confidential knowledge. This last aspect seems especially relevant, as R is used frequently in these organizations.

Resource Restrictions

The other category of problems is not so much related to intentional abuse, and might even arise unintentionally. It is fair to say that R can be quite greedy with hardware resources. One can easily run a command which will consume all of the available memory and/or CPU, and does not finish executing unless manually terminated. When running R on the local machine through the interactive console, the user will quickly recognize a function call that is not returning timely, and can interrupt the process prematurely by sending a `SIGINT` by pressing

CTRL+C in Linux or ESC in Windows. If this doesn't work we can open the task manager and tell the operating system to kill the process.

However, when R is embedded in a bigger system, things are more complicated, and we have to think about these scenarios in advance. When an out-of-control R job is not properly detected, the process might very well run forever and take down our service, or even the entire machine. This has actually been a major problem that we personally experienced in an early implementation of a public web service for mixed modelling (Ooms 2010) which uses the lme4 package (Bates, Maechler, and Bolker 2011). What happened was that users could accidentally specify a variable with many levels as the *grouping factor* which would cause the design matrix to blow up, even on a relatively small dataset, and decompositions would take forever to complete. To make things worse, lme4 uses a lot of C code which does not respond do time limits set by R's setTimeLimit function. This happened multiple times, and the only way to get things up again was to manually login to the server and reset the application.

This example is not an exception. The behavior of R can sometimes be unpredictable, which is an aspect that is easily overlooked by (non-statistician) developers. When a system calls out to e.g. an SQL or php script, the script usually runs without any problems and the time needed to process is proportional to the size of the data, i.e. the number of returned records returned by SQL. However, in an R script, many things can go wrong, even though the script itself is perfectly fine. Algorithms might not converge, data might be rank-deficient, or missing values throw a spanner in the works. Even when we only use tested code or predefined services, this does not always entirely guarantee smooth and timely completion of R jobs. When using R in systems or shared facilities, it is important that we take this aspect into account and have a way of dealing with this that does not require manual intervention.

3. Different Approaches of Restricting R

The current section introduces some approaches of securing and sandboxing R, with their advantages and limitations. They are reviewed in the context of our use cases, and evaluated on how they address the problems of system abuse and resource restrictions. The approaches are increasingly *low-level*: they represent security on the level of the application, R software itself and operating system. As will become clear, we are leaning towards the opinion that R itself is not very well suited to address security issues, and the only way to do proper sandboxing is on the level of the operating system. This will lead us to the introduction of the **RAppArmor** package, which is described in the next section.

3.1. Application Level Security: Predefined Services

The most common approach to preventing system abuse is simply to only allow a limited set of predefined services, that have been deployed by a trusted developer and cannot be abused. This is generally the case in websites containing dynamic content though e.g. CGI or PHP scripts. Running arbitrary code is explicitly prevented and any possibility to do so anyway is considered a security hole. For example, we might want to expose the following function as a web service:

```
liveplot <- function (ticker) {
  url <- paste("http://ichart.finance.yahoo.com/table.csv?s=",
```

```
    ticker, "&a=07&b=19&c=2004&d=07&e=13&f=2020&g=d&ignore=.csv",
    sep = "")
  mydata <- read.csv(url)
  mydata$Date <- as.Date(mydata$Date)
  myplot <- ggplot2::qplot(Date, Close, data = mydata, geom = c("line",
    "smooth"), main = ticker)
  print(myplot)
}
```

This function above downloads live data from the public API at Yahoo Finance and creates an on-demand plot of the historical values using `ggplot2` (Wickham 2009). The function has only one parameter, `ticker`, which is a character string identifying a stock symbol. This function can be exposed as a predefined web service, where the client only supplies the `ticker` argument. Hence the system does not need to run any potentially harmful user-supplied R code. The client can only set the symbol to e.g. `'GOOG'` and the resulting plot can be returned in the form of a PNG image or PDF document. This function is actually the basis of the "stockplot" web application (Ooms 2009); an interactive graphical web application for financial analysis which still runs today.

Limiting users or clients to execute only predefined services is often the easiest solution, but rather limited in application and actually not 100% secure. A predefined service can be nice to do some canned calculations or generate a plot as done in the example, but for most R applications it quickly turns out to be overly restrictive. For example in case of an application that allows the user to fit a statistical model, the user might need to be able to include transformations of variables like `I(cos(x^ 2))` or `cs(x, 3)`. Not allowing a user to call any custom functions makes this hard to implement. Furthermore, when using only predefined services, all the work and responsibility is put in the hands of the developer and administrator. Only they can expose new services and they have to make sure that all services that are exposed cannot be abused in some way or another. Therefore this approach is expensive, and not very social in terms of users contributing additional services. In practice, anyone that wants to publish an R service will have to purchase and manage a personal server or know someone that is willing to do so.

Also it might still be necessary to set hardware limitations, even when exposing relatively simple, restricted services. We already mentioned the example of the `lme4` web application, where a single user could accidentally take down the entire system by specifying an overly complex model. Hence, restricting to predefined services does not quite guarantee smooth and timely completion of R jobs.

Code Injection

Finally, there is still the risk of *code injection*. Because R is a very dynamic language, evaluations sometimes happen at unexpected places. One example is during the parsing of `formulas`. For example, we might want to publish a service that calls the `lm()` function in R on a fixed dataset. Hence the only thing the user can supply is a *formula* in the form of a character vector. Assume in the code snippet below that the `userformula` is a string that has been supplied by a user through some graphical interface.

```
glm(userformula, data=cars)
```

For example the user might supply a string `"speed ~ dist"` and the service will return the coefficients. On first sight, this might seem like a safe service. However, formulas actually allow for the inclusion of calls to other functions. So even though the `userformula` is a character vector, we can actually use it to inject a function call:

```
userformula <- "speed ~ dist + system('whoami')"
lm(userformula, data=cars)
```

In the example above, `lm` will automatically convert `userformula` from type character to a `formula`, and subsequently execute the `system("whoami")` command. So even when a user can supply only very simple primitive data, it is still important to sanitize the input before calling the service. One way to do so is to set up the service in such a way that only alphanumeric values are needed for the parameters, and use a regular expression to remove any other characters, before actually executing the script or service:

```
myarg <- gsub("[^a-zA-Z1-9]", "", myarg)
```

3.2. Sanitizing and Blacklisting

A less restrictive approach is to allow users to push custom R code, but inspect the code before evaluating it to make sure it does not contain malicious calls. This approach has been adopted with some web sites that allow users to run R code, like Banfield (1999) and Cloudstat (2012). However, given the dynamic nature of R, this is actually very hard to do and is often easy to circumvent. For example, one might want to prevent users from calling the `system` function. One way is to define some smart regular expressions that look for the word "system" in a block of code. This way it would be possible to detect a potentially malicious call like this:

```
system("whoami")
```

However, it will be much harder to detect the equivalent call in the following block:

```
foo <- get(paste("sy", "em", sep="st"))
bar <- paste("who", "i", sep="am")
foo(bar)
```

And indeed, it turns out that the services that use this approach are fairly easy to hack. Because R is a dynamic scripting language, the exact function calls might not reveal themselves until runtime, when it is often too late. We are actually quite convinced that it is nearly impossible to really sanitize an R script just by inspecting the source code.

An alternative method to do sanitizing is to define an extensive whitelist of functions that a user is allowed to call, and mask all other functions. The **sandboxR** (Daroczi 2012) package uses this method to block access to all R functions that provide access to the file system. It evaluates the user-supplied code in an environment in which all blacklisted functions are masked from the calling namespace. This is fairly effective and can be useful for some applications. However, the method relies on exactly knowing and specifying which functions are

safe and which are not. The package author has done this for the thousands of R functions in the base package and we assume he has done a good job. However, it makes it hard to maintain and cumbersome to generalize the approach to other R packages (by default the method does not allow loading other packages). Furthermore the entire method falls if there is one function that has been overlooked, which does make the method somewhat vulnerable.

Moreover, even when sanitizing of the code is successful, this method does not limit the use of hardware resources in any way. Hence, additional methods are still required to prevent excessive use of resources in a public environment.

3.3. Sandboxing on the Level of the Opering System

One can argue that managing hardware and security privileges is something that is outside the domain of the R software, and is better let to the operating system. The R software has been designed for statistical computing and related functionality; the operating system deals with hardware and security related matters. Hence, in order to really sandbox R properly without imposing unnecessary limitations on its functionality, we need to sandbox the *R-process* on a lower level in the OS. When restrictions are enforced by the operating system instead of R itself, we do not have to worry about all of the pitfalls and implementation details of R. The user can interact freely with R, but won't be able to do anything for which the system does not grant permissions. Unfortunately, this approach comes at the cost of portability of the software. Different operating systems implement very different methods for managing processes and privileges, so the solutions will be to a large extend OS-specific. However we can still create interfaces from R to interact directly with the operating system. And as is often the case in R, eventually these functions can behave somewhat OS specific, providing similar functionality on different systems and abstract away low level technicalities.

Some operating systems offer more advanced capabilities for setting process restrictions than others. The most advanced functionality is found in `UNIX` like systems, of which the most popular ones are either `BSD` based (`FreeBSD, OSX,` etc) or `Linux` Based (`Debian, Ubuntu, Fedora, Redhat, Suse,` etc). Most `UNIX` like systems implement some sort of `ULIMIT` functionality to facilitate restricting availability of hardware resources on a per-process basis. Furthermore, on both `BSD` and `Linux` there are a number of *Mandatory Access Control* (`MAC`) systems available. On Linux, these are offered as Kernel modules. The most popular ones are `AppArmor` (Bauer 2006), `SELinux` (Smalley, Vance, and Salamon 2001) and `Tomoyo Linux` (Harada, Horie, and Tanaka 2004). MAC provides a much finer degree of control than standard user-based privileges, by applying advanced security policies on a per-process basis.

Using a combination of `MAC` and `ULIMIT` tools we can do a pretty decent job in sandboxing a single R process to a point where it can do little harm to the system. Hence we can run arbitrary R code without losing any sleep over potentially jeopardizing the machine. In the next section a reference implementation is presented based on the `Linux` and `AppArmor`.

4. the RAppArmor package

The current section describes some security concepts and how an R process can be sandboxed using a combination of `ULIMIT` and `MAC` tools. The methods are illustrated using the **RAppArmor** package: an implementation based on `Linux` and `AppArmor`. AppArmor ("Application Armor") is a security module for the Linux kernel. It allows the system administrator to asso-

ciate *security profiles* to programs and processes that restrict the capabilities and permissions of that process. The **RAppArmor** R package implements some convenient R interfaces to Linux system calls related to setting privileges and restrictions of a running process. Besides applying AppArmor profiles, **RAppArmor** also interfaces to the Linux `prlimit` call, which sets `RLIMIT` (resource limit) values on a process (`RLIMIT` are the linux implementation of `ULIMIT`). Linux defines a number of `RLIMIT`'s, which define resources like memory, number of processes, and stack size. More on `RLIMIT` later in section 4.6.

There are two ways of using AppArmor. One is to automatically associate a single security profile with every R process. This can also be done without **RAppArmor**. However, this is often somewhat limited and overly restrictive. Using the RAppArmor package, R commands can be dynamically evaluated in a secured sandbox with a custom uid, security profile and hardware restrictions, with minimal overhead. This gives the user the freedom to design code in which only the insecure parts are sandboxed, which turns out to be quite powerful.

4.1. AppArmor Profiles

The security profiles are the core of the AppArmor software. A profile is defined by a set of rules in a text file using AppArmor syntax. The Linux kernel translates these rules to a security policy that it will enforce on the appropriate process. A brief introduction to the AppArmor syntax is given in section 5.1. The appendix of this paper contains some example profiles that ship with the **RAppArmor** package to get the user started. When the package is installed through the Ubuntu installer (e.g. using apt-get) the profiles are automatically copied to `/etc/apparmor.d/rapparmor.d/`. Because profiles define file access permissions based the location of files and directories on the file system, they are to some extend specific to a certain Linux distribution, as different distributions have somewhat varying conventions on where files are located. The example profiles included with **RAppArmor** are based on the file layout of the **r-base** package (and its dependencies) by Bates and Eddelbuettel (2004) for Debian/Ubuntu, currently maintained by Dirk Eddelbuettel.

The `RAppArmor` package and the included profiles work "out of the box" on Ubuntu 12.04 (Precise) and up. Also it should be working on Debian 7.0 (Wheezy) and up, however as of writing of this paper, this distribution is still in the "testing" phase. Furthermore the package has been successfully build on OpenSuse 12.1. Note that Suse systems organize the file system in a slightly different way than Ubuntu and Debian, so the profiles should be modified accordingly.

Again, we want to emphasize that the package and included profiles should mostly be seen as a *reference implementation*. Using the package we demonstrate how to create a working sandbox using AppArmor. However, depending on system and application, different policies might be appropriate. The **RAppArmor** package provides the tools to set security restrictions in R and ships with some example profiles to get the user started. However it is still up to the administrator to determine which security policy is appropriate for a certain system and context. The example profiles are a good starting point, but should be fine-tuned for specific applications.

4.2. Automatic Installation

The **RAppArmor** package consists of an R package and a number of security profiles. On Ubuntu Linux the package is most easily installed using the binary packages provided through

launchpad:

```
sudo add-apt-repository ppa:opencpu/rapparmor
sudo apt-get update
sudo apt-get install r-cran-rapparmor
```

One can also create the Ubuntu packages from the source R package using something along the lines of the following:

```
tar xzvf RAppArmor_0.5.0.tar.gz
cd RAppArmor/
debuild -uc -us
cd ..
sudo dpkg -i r-cran-rapparmor_0.5.0-precise0_amd64.deb
```

The `r-cran-rapparmor` Ubuntu package will automatically install required dependencies and security profiles. The security profiles are installed in `/etc/appamor.d/rapparmor.d/`.

4.3. Manual Installation

To install the package on a distribution for which no installation package is available, one might need a manual installation. To build the package manually several steps are needed. First of all, one needs to make sure the required dependencies are installed:

```
sudo apt-get install r-base-dev libapparmor-dev apparmor
```

Note that the package requires R version 2.14 or higher. Also the system needs to have an apparmor enabled Linux kernel. After these packages are installed, one can proceed installing **RAppArmor** in R, using e.g:

```
sudo R CMD INSTALL RAppArmor_0.5.0.tar.gz
```

This will compile the C code and install the R package. After the package has been installed successfully, the security profiles need to be copied to the `apparmor.d` directory:

```
cd /usr/local/lib/R/site-library/RAppArmor/
sudo cp -Rf profiles/debian/* /etc/apparmor.d/
```

Finally, the apparmor service needs to be restarted to load the new profiles. Also we do not want to enforce default the R profiles at this point yet:

```
sudo service apparmor restart
sudo aa-disable usr.bin.r
```

This should complete the installation. To verify if everything is working, start R and run the following code:

```
library(RAppArmor)
aa_change_profile("r-base")
```

If the code runs without any errors, the package has successfully been installed.

4.4. Linux Security Methods

The **RAppArmor** package interfaces to a number of Linux system calls that are useful in the context of security and sandboxing. The advantage of calling these directly from R is that we can dynamically set the parameters from within the R process, as opposed to fixing them for every R session. Hence it is actually possible to execute some parts of an application in a different security context other parts, which can be useful in large applications.

The package defines a lot of low level functions that wrap around Linux C interfaces. However it is not required to study all of these functions. For the end user, everything in the package comes together in the powerful and convenient `eval.secure()` function. This function mimics `eval()`, but it has additional parameters that define restrictions which will be enforced to this specific evaluation. For example, one could use

```
myresult <- eval.secure(myfun(), RLIMIT_AS = 10*1024*1024, profile="r-base")
```

Which will call `myfun()` with a memory limit of 10MB and the "r-base" security profile (which is introduced in section A.1). The `eval.secure` function works by creating a *fork* of the current process, and then set hard limits, UID and apparmor profile on the forked process, before evaluating the call. After the function returns, or when the timeout is reached, the forked process is killed and cleaned up. This way, all of the one-way security restrictions can be applied, and evaluations that happen inside `eval.secure` won't have any side effects on the main process.

4.5. Setting User and Group ID

One of the most basic security methods is running a process as a specific user. Especially within a system where the main process has superuser privileges (which could be the case in for example a webserver), switching to a user with limited privileges before evaluating any code is a wise thing to do. We could even consider a design where every user of the application has a dedicated user account on the Linux machine. The **RAppArmor** package implements the functions `getuid, setuid, getgid, setgid`, which call out to the respective `Linux` system calls. Users and groups are defined as integer values as specified inside the `/etc/passwd` file.

```
> library(RAppArmor)
> system('whoami')
root
> getuid()
[1] 0
> getgid()
[1] 0
> setgid(1000)
Setting gid...
```

```
> setuid(1000)
Setting uid...
> getgid()
[1] 1000
> getuid()
[1] 1000
> system('whoami')
jeroen
```

The user/group ID can also be set inside the `eval.secure` function. In this case it will not affect the main process; the UID is only set for the time of the secure evaluation.

```
> eval(system('whoami', intern=TRUE))
[1] "root"
> eval.secure(system('whoami', intern=TRUE), uid=1000)
[1] "jeroen"
> eval(system('whoami', intern=TRUE))
[1] "root"
```

Note that in order for `setgid` and `setuid` to work, the user must have the appropriate capabilities in `Linux`, which are usually restricted to users with superuser privileges. The `getuid` and `getgid` functions can be called by anyone.

4.6. Linux Resource Limits (RLIMIT)

Linux defines a number of `RLIMIT` values that can be used to set resource limits on a process (Free Software Foundation 2012). The **RAppArmor** package has functions to get/set to the following RLIMITs:

- `RLIMIT_AS` – The maximum size of the process's virtual memory (address space).
- `RLIMIT_CORE` – Maximum size of core file.
- `RLIMIT_CPU` – CPU time limit.
- `RLIMIT_DATA` – The maximum size of the process's data segment.
- `RLIMIT_FSIZE` – The maximum size of files that the process may create.
- `RLIMIT_MEMLOCK` – Number of memory that may be locked into RAM.
- `RLIMIT_MSGQUEUE` – Max number of bytes that can be allocated for POSIX message queues
- `RLIMIT_NICE` – Specifies a ceiling to which the process's nice value (priority).
- `RLIMIT_NOFILE` – Limit maximum file descriptor number that can be opened.
- `RLIMIT_NPROC` – Maximum number of processes (or, more precisely on Linux, threads) that can be created by the user of the calling process.

- `RLIMIT_RTPRIO` – Ceiling on the real-time priority that may be set for this process.
- `RLIMIT_RTTIME` – Limit on the amount of CPU time that a process scheduled under a real-time scheduling policy may consume without making a blocking system call.
- `RLIMIT_SIGPENDING` – Limit on the number of signals that may be queued by the user of the calling process.
- `RLIMIT_STACK` – The maximum size of the process stack.

For all of the above `RLIMITs`, the **RAppArmor** package implements a function which name is equivalent to the non-capitalized name of the `RLIMIT`. For example to get/set `RLIMIT_AS`, one can call `rlimit_as()`. Every `rlimit_` function has exactly 3 parameters: `hardlim`, `softlim`, and `pid`. Each argument is specified as an integer value. The `pid` arguments points to the target process. When this argument is omitted, the calling process is targeted. When the `softlim` is omitted, it is set equal to the `hardlim`.

The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may only set its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the `CAP_SYS_RESOURCE` capability) may make arbitrary changes to either limit value. When the function is called without any arguments, it prints the current limits to `STDOUT`. (Free Software Foundation 2012)

```
> library(RAppArmor)
> rlimit_as()
RLIMIT_AS:
Current limits: soft=-1; hard=-1
> A <- rnorm(1e7)
> rm(A)
> gc()
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 185467  5.0     407500 10.9   350000  9.4
Vcells 176590  1.4    8743611 66.8 10182143 77.7
>
> rlimit_as(10*1024*1024)
RLIMIT_AS:
Previous limits: soft=-1; hard=-1
Current limits: soft=10485760; hard=10485760
> A <- rnorm(1e7)
Error: cannot allocate vector of size 76.3 Mb
```

Note that a process owned by a user without superuser privileges can only modify `RLIMIT` to more restrictive values. However, using `eval.secure`, a more restrictive `RLIMIT` can be applied to a single evaluation without any side effects on the main process:

```
> library(RAppArmor)
> A <- eval.secure(rnorm(1e7), RLIMIT_AS = 10*1024*1024);
Error: cannot allocate vector of size 76.3 Mb
> A <- rnorm(1e7)
```

The exact meaning of the different limits can be found in the `RAppArmor` package documentation (e.g. `?rlimit_as`) or in the documentation of the Linux operating system (Canonical, Inc 2012).

4.7. Activating AppArmor profiles

The **RAppArmor** package implements three calls to the Linux kernel related to applying AppArmor profiles: `aa_change_profile`, `aa_change_hat` and `aa_revert_hat`. Both the `aa_change_profile` and `aa_change_hat` functions take a parameter named `profile`: a character string identifying the name of the profile. This profile has to be preloaded by the kernel, before it can be applied to a process. The easiest way to load profiles is to copy them to the directory `/etc/apparmor.d` and then run `sudo service apparmor restart`.

The main difference between a *profile* and a *hat* is that switching profiles is an irreversible action. Once the profile has been associated with the current process, the process cannot call `aa_change_profile` again to escape from the profile (that would defeat the purpose). The only exception to this rule are profiles that contain an explicit `change_profile` directive. The `aa_change_hat` function on the other hand is designed to associate a process with a security profile in a way that does allows escaping out of the security profile. In order to realize this, the `aa_change_hat` takes a second argument called *magic_token*, which defines a secret key that can be used to *revert* the hat. When `aa_revert_hat` is called with the same `magic_token` that was used in `aa_change_hat`, the security restrictions are relieved. This can be very useful in certain applications, however it is also a security risk. It is important that the code which is running in the sandbox cannot read the magic token from memory somehow, because then it would be have a way of breaking out of the sandbox. Hence, storing the `magic_token` in a variable that is readable from within the sandbox might not be a good idea.

The **RAppArmor** package ships with a profile called *testprofile* which contains a hat called *testhat*. We use this profile to demonstrate the functionality. The profiles have been defined such that *testprofile* allows access to `/etc/group` but denies access to `/etc/passwd`. The *testhat* denies access to both `/etc/passwd` and `/etc/group`.

```
> library(RAppArmor);
> result <- read.table("/etc/passwd")

> aa_change_profile("testprofile")
Switching profiles...
> passwd <- read.table("/etc/passwd")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") : cannot open file '/etc/passwd': Permission denied
> group <- read.table("/etc/group")

> mytoken <- 13337;
> aa_change_hat("testhat", mytoken);
Setting Apparmor Hat...

> passwd <- read.table("/etc/passwd")
```

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") : cannot open file '/etc/passwd': Permission denied
> group <- read.table("/etc/group")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") : cannot open file '/etc/group': Permission denied

> aa_revert_hat(mytoken);
Reverting AppArmor Hat...

> passwd <- read.table("/etc/passwd")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") : cannot open file '/etc/passwd': Permission denied
> group <- read.table("/etc/group")
```

Just like for `setuid` and `rlimit` functions, `eval.secure` can be used to enforce an AppArmor security profile on a single call, witout any side effects. The `eval.secure` function uses `aa_change_profile` and is therefore most secure.

```
> out <- eval(read.table("/etc/passwd"))
> nrow(out)
[1] 68
> out <- eval.secure(read.table("/etc/passwd"), profile="testprofile")
Error in file(file, "rt") : cannot open the connection
```

4.8. AppArmor without RAppArmor

The **RAppArmor** package allows us to dynamically load an AppArmor profile from within an R session. This gives a great deal of flexability. However, it is also possible to use AppArmor without the **RAppArmor** package, by setting a single profile to be loaded with any running R process.

The RAppArmor package ships with a file named `usr.bin.r`. At the installation of the package, this file is copied to `/etc/apparmor.d/`. This file is basically a copy of the `r-user` profile in appendix A.3, however with a small change: where `r-user` defines a named profile with

```
profile r-user {
  ...
}
```

the `usr.bin.r` file defines a profile specific to a filepath:

```
/usr/bin/R {
  ...
 }
```

When using the latter syntax, the profile is automatically associated every time the file `/usr/bin/R` is executed (which is the file that runs when `R` is entered in the shell). This way we can set some default security restrictions for our daily work. Profiles tied to a specific program can be activated by the administrator using:

```
sudo aa-enforce usr.bin.r
```

This will enforce the security restrictions on every new R process that is started. To disable the profile, the administrator can run:

```
sudo aa-disable usr.bin.r
```

After disabling the profile, the R program can be started without any restrictions.

Note that the `usr.bin.r` profile does **not** grant permission to change profiles. Hence, one the `usr.bin.r` profile is in enforce mode, we cannot use the `eval.secure` or `aa_change_profile` functions from the **RAppArmor** package to change into a different profile, as this would be a security hole.

4.9. Learning using Complain Mode

Finally AppArmor allows the administrator to set profiles in *complain mode*, which is also called *learning mode*.

```
sudo aa-complain usr.bin.r
```

This is useful for developing new profiles. When a profile is set in complain mode, security restrictions are not actually enforced; instead all violations of the security policy are logged to the `syslog` and `kern.log` files. This is a powerful way of creaing new profiles: one can set a program in complain mode during regular use, and afterwards the log files can be used to study violations of the current policy. From these violations we can determine which permissions will have to be added to the profile to make the program work under normal behavior. AppArmor even ships with a powerful utility named `aa-logprof` which helps the administrator by parsing log files and suggesting new rules to be added to the profile. This is a nice way of debugging a profile, and figuring out which permissions exactly a program requires to do its work.

5. Profiling R: Defining Security Policies

The "hard" part of the problem is actually profiling R. With profiling we mean defining the policies: which files and directories should R be allowed to read and write to? Which external programs is it allowed to execute? Which libraries or shared modules it allowed to load, etc. We want to minimize ways in which the process could potentially damage the system, but we don't want to be overly restrictive either: preferebly, users should be able to do anything they normally do in R. Because R is such a complete system with a big codebase and a wide range of functionality, the base system actually already requires quite a lot of access to the file system.

As often, there is no "one size fits all" solution. Depending on which functionality is needed for an application we might want to grant or deny certain privileges. We might even want to

execute some parts of a process with tighter privileges than other parts. For example, within a web service, the service process should be able to write to system log files, which should not be writable by custom code from a user. We might also want to be more strict on some users than others, e.g. allow all users to execute code, but only allow privileged users to install a new package.

5.1. AppArmor Policy Configuration Syntax

The *AppArmor policy configuration syntax* is used to define the access control profiles in `AppArmor`. Other mandatory access control systems might implement different functionality and require other syntax, but in the end they address mostly similar issues. AppArmor is a quite advanced and provides access control over many of the features and resources found in the Linux kernel, e.g. file access, network rules, Linux capability modes, mounting rules, etc. All of these can be useful, but most of them are very application specific. Furthermore, the policy syntax has some additional functionality that allows for defining *subprofiles*, and *includes*.

The most important form of access control which will be the focus of the remaining of the section are *file permission access modes*. Once AppArmor is enforcing mandatory access control, a process can only access files and directories on the system for which it has explicitly been granted access in its security profile. Because in `Linux` almost everything is a file (even sockets, devices, etc) this gives a great deal of control. AppArmor defines a number of access modes on files and directories, of which the most important ones are:

`r` – read file or directory

`w` – write to file or directory

`m` – load file in memory

`px` – discrete profile execute of executable file

`cs` – transition to subprofile for executing a file

`ix` – inherit current profile for executing a file

`ux` – unconfined execution of executable file (dangerous)

Using this syntax we will present some example profiles for R. Because the profile depends on the absolute locations of system files, we will assume the standard file layout for Debian and Ubuntu systems. This includes files that are part of `r-base` and other packages that are used by R, e.g. `texlive`, `libxml2`, `bash`, `libpango`, `libcairo`, etc.

5.2. Profile: `r-base`

Appendix A.1 contains a profile that we have named `r-base`. It is a fairly basic and general profile. It grants read/load access to all files in common shared system directories, e.g. `/usr/lib, /usr/local/lib, /usr/share`, etc. However, the default profile only grants write access inside `/tmp`, not in e.g. the home directory. Furhermore, R is allowed to execute any of the shell commands in `/bin` or `/usr/bin` for which the program will inherit the restrictions.

```
> library(RAppArmor)
> aa_change_profile("r-base")
Switching profiles...

> #These operations will be denied:
> list.files("/")
character(0)
> list.files("~")
character(0)
> file.create("~/test")
[1] FALSE
> list.files("/tmp")
character(0)
> install.packages("wordcloud")
Error opening file for reading: Permission denied

> #These operations are permitted:
> library(ggplot2);
> setwd(tempdir())
> pdf("test.pdf")
> qplot(speed, dist, data=cars);
> dev.off()
null device
          1
> list.files()
[1] "downloaded_packages"
[2] "libloc_107_669a3e12.rds"
[3] "libloc_118_46fd5f8e.rds"
[4] "libloc_128_97f33314.rds"
[5] "pdf6d1117f7d683"
[6] "repos_http%3a%2f%2fcran.stat.ucla.edu%2fsrc%2fcontrib.rds"
[7] "test.pdf"
> file.remove("test.pdf")
[1] TRUE
```

The `r-base` profile effectively prevents R from most malicious activity, while still allowing access to all of the libraries, fonts, icons, and programs that it might want to use. One thing to note is that the profile does not allow listing of the contents of `/tmp`, but it does allow full rw access on any of its subdirectories. This is to prevent one process from reading/writing files from the temp directory of another active R process (given that it cannot discover the name of the other temp directory).

The `r-base` profile is a quite liberal and general purpose profile. When using AppArmor in a more specific application, it is recommended to make the profile a bit more restrictive by specifying exactly *which* of the packages, shell commands and system libraries should be accessible by the application. That could prevent potential problems when vulnerabilities are found in some of the standard libraries.

5.3. Profile: `r-compile`

The `base-r` profile does not allow access to the compiler, nor does it allow for loading (`m`) or execution (`ix`) of files in places where it can also write. If we want user to be able to compile e.g. C++ code, we will need to give it access to the compiler. In order to do so, we need to add these lines:

```
/usr/include/** r,
/usr/lib/gcc/** rix,
/tmp/** rmw,
```

Note especially the last line. The `m` allows R to load shared objects into memory from anywhere under `/tmp`. This is needed to load the compiled code after it has been installed to a temporary directory. Note that this does not come without a cost: compiled code can potentially contain malicious code or even exploits that can do harm when loaded into memory. If this privilege is not needed, it is generally recommended to only allow `m` and ix access modes on files that have been installed by the system administrator. The new profile including these rules ships with the package as `r-compile` and is also printed in appendix A.2.

After adding the lines above and reloading the profile, it should be possible to compile a package that contains C++ code and install it to somewhere in `/tmp`:

```
> eval.secure(install.packages("wordcloud", lib=tempdir()), profile="r-compile");
trying URL 'http://cran.stat.ucla.edu/src/contrib/wordcloud_2.0.tar.gz'
downloaded 36 Kb

* installing *source* package 'wordcloud' ...
** package 'wordcloud' successfully unpacked and MD5 sums checked
** libs
g++ -I/usr/share/R/include -DNDEBUG -I"/usr/local/lib/R/site-library/Rcpp/include"
   -fpic  -O3 -pipe  -g  -c layout.cpp -o layout.o
g++ -shared -o wordcloud.so layout.o -L/usr/local/lib/R/site-library/Rcpp/lib
   -lRcpp -Wl,-rpath,/usr/local/lib/R/site-library/Rcpp/lib -L/usr/lib/R/lib -lR
installing to /tmp/RtmpFCM6WS/wordcloud/libs
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded

* DONE (wordcloud)

The downloaded source packages are in
        '/tmp/RtmpFCM6WS/downloaded_packages'
```

5.4. Profile: r-user

Appendix A.3 defines a profile named r-user. This profile is designed to be a nice balance between security and freedom for day to day use of R. It extends the r-compile profile with some additional privileges with respect to the users home directory. The variable @{HOME} is defined in the tunables/global include and matches the location of the users home directory, i.e. /home/jeroen for a user named "jeroen". The profile assumes that there is a directory named R directly inside the home directory (e.g /home/jeroen/R), to which R can read and write. Furthermore, R can load and execute files in the directories i686-pc-linux-gnu-library and x86_64-pc-linux-gnu-library inside of this R directory. These are the standard locations where R installs a users personal package library.

Using the r-user profile, the user will be able to do most of his day to day work, including installing and loading new packages in his personal library, while still being protected against most malicious activities. The r-user profile is also the basis of the default usr.bin.r profile as described in section 4.8.

5.5. Installing packages

An additional privilege that might be needed in some situations is the option to install packages to the machines global library, which is readable by all users. In order to allow this, a profile needs to include write access to the site-library directory:

```
/usr/local/lib/R/site-library rmw,
```

After adding this line to a profile, the policy will allow for installing R packages to the global site library. However, note that AppArmor does not replace, but *supplements* the standard access control system. Hence if a user does not have superuser privileges it will still not be able to install packages in the global site library, even though the AppArmor profile does grant this permission.

6. Concluding remarks

In this paper the reader was introduced to some potential security issues related to the use of the R software. We hope to have raised awareness that security is an increasingly important concern for the R user, but also that addressing this issue could open doors to new applications of the software. The **RAppArmor** package was introduced as an example that demonstrates how some security issues could be addressed using facilities from the operating system, in this case `Linux`. This implementation provides a starting point for creating a sandbox in R, but as was emphasized throughout the paper, it is still up to the administrator to actually design security policies that are appropriate for a certain application or system.

Our package uses the `AppArmor` software from the `Linux` kernel, which works for us, but this is just one way to approach the issue. `Linux` has two other mandatory access control systems that are worth exploring: `TOMOYO` and `SELinux`. Especially the latter is known to be very sophisticated, but also extremely hard to set up. Other more recent technology that might be interesting is provided by `Linux CGroups`. Using `CGroups`, control of allocation and security can be managed using hierarchical process groups. The upcoming `LXC` (Linux Containers) build on top of `CGroups` to provide virtual environments which have their own process and network space. A completely different direction is suggested by `renjin` (Bertram 2012), a `JVM`-based Interpreter for the R Language. If R code can be executed though the `JVM`, we might be able to use some tools from the `Java` community to address similar issues. Finally we curious to see what the `BSD` community has to offer in this area, as `BSD` distributions are known to have a lot of emphasis on security in the design of the operating system.

However, no matter which tools are used, security always comes down to the trade off between *use* and *abuse*. This has a major social aspect to it, and is a learning process in itself. A balance has to be found between providing enough freedom to use facilities as desired, yet avoid undesired activity. Apart from the technical parameters, a good balance also depends on factors like what exactly is considered undesired and how well you know your users. For example a job using 20 parallel cores might be considered as abusive by many administrators, but might actually be regular use for a MCMC simulation study on a supercomputer. Security policies are not unlike legal policies in the sense that creating a good one is an iterative process. It might not be until you actually put an application in production that you start getting complains from users that their favorite package is not working, or that you find out that some users are abusing the system in a way that was hard to foresee.

A. Example Profiles

This appendix prints some of the example profiles that ship with the **RAppArmor** package. To load them in AppArmor, simply copy-paste the profile into a file that you put in the directory `/etc/apparmor.d` and then run `sudo service apparmor restart`. You should then be able to load them into an R session using either the `aa_change_profile` or `secure.eval` function from the `RAppArmor` package.

A.1. Profile: `r-base`

```
#include <tunables/global>
profile r-base {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/resolv.conf r,
        /etc/xml/* r,
        /tmp/** rw,
        /usr/bin/* rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
        /usr/share/ca-certificates/** r,
}
```

A.2. Profile: `r-compile`

```
#include <tunables/global>
profile r-compile {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/resolv.conf r,
        /etc/xml/* r,
```

```
        /tmp/** rmw,
        /usr/bin/* rix,
        /usr/include/** r,
        /usr/lib/gcc/** rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
        /usr/share/ca-certificates/** r,
}
```

A.3. Profile: r-user

```
#include <tunables/global>
profile r-user {
        #include <abstractions/base>
        #include <abstractions/nameservice>

        capability kill,
        capability net_bind_service,
        capability sys_tty_config,

        @{HOME}/ r,
        @{HOME}/R/ r,
        @{HOME}/R/** rw,
        @{HOME}/R/{i686,x86_64}-pc-linux-gnu-library/** mrwix,
        /bin/* rix,
        /etc/R/ r,
        /etc/R/* r,
        /etc/fonts/** mr,
        /etc/resolv.conf r,
        /etc/xml/* r,
        /tmp/** mrwix,
        /usr/bin/* rix,
        /usr/include/** r,
        /usr/lib/gcc/** rix,
        /usr/lib/R/bin/* rix,
        /usr/lib{,32,64}/** mr,
        /usr/lib{,32,64}/R/bin/exec/R rix,
        /usr/local/lib/R/** mr,
        /usr/local/share/** mr,
        /usr/share/** mr,
        /usr/share/ca-certificates/** r,
}
```

B. Security Unit Tests

This appendix prints a number of unit tests that contain malicious code and which should be prevented by any sandboxing tool.

B.1. Access System Files

Usually R has no business in the system logs, and these are not included in the profiles. The codechunk below attempts to read the syslog file.

```
readSyslog <- function(){
        readLines('/var/log/syslog');
}
```

When executing this with a r-user profile, access to this file is denied, resulting in an error:

```
> eval.secure(readSyslog(), profile='r-user')
Switching profiles...
Error in file(con, "r") : cannot open the connection
```

B.2. Access User Files

Access to system files can to some extend by prevented by running processes as non privileged users. But it is easy to forget that also the users personal files can contain senstive information. Below a simple function that scans the `Documents` directory of the current user for files that contain credit card numbers.

```
findCreditCards <- function(){
  pattern <- "([0-9]{4}[- ]){3}[0-9]{4}"
  for (filename in list.files("~/Documents", full.names=TRUE, recursive=TRUE)){
    if(file.info(filename)$size > 1e6) next;
    doc <- readLines(filename)
    results <- gregexpr(pattern, doc)
    output <- unlist(regmatches(doc, results));
    if(length(output) > 0){
      cat(paste(filename, ":", output, collapse="\n"), "\n")
    }
  }
}
```

This example prints the credit card numbers to the user, but it would be quite easy to post them to some server on the internet. For this reason the `r-user` profile denies access to the users home dir, except for the `~/R` directory.

B.3. Limit Memory

When a system or service is used by many users at the same time, it is important that we cap the memory that can be used by a single process. The following function generates a quite large matrix:

```
memtest <- function(){
        A <- matrix(rnorm(1e7), 1e4);
}
```

When R tries to allocate more memory than allowed, it will throw an error:

```
> A <- eval.secure(memtest(), RLIMIT_AS = 1000*1024*1024)
RLIMIT_AS:
Previous limits: soft=-1; hard=-1
Current limits: soft=1048576000; hard=1048576000
> rm(A)
> gc()
         used (Mb) gc trigger  (Mb) max used  (Mb)
Ncells 193074 10.4     407500  21.8   350000  18.7
Vcells 299822  2.3   17248096 131.6 20301001 154.9


> A <- eval.secure(memtest(), RLIMIT_AS = 100*1024*1024)
RLIMIT_AS:
Previous limits: soft=-1; hard=-1
Current limits: soft=104857600; hard=104857600
Error: cannot allocate vector of size 76.3 Mb
```

B.4. Limit CPU Time

Suppose we are hosting a web service and we want to kill jobs that do not finish in 5 seconds. Below a snippet that will take much more than 5 seconds on most machines. Note that because R calling out to `C` code, it will not be possible to terminate this function prematurely using R's `setTimeLimit` or even using `CTRL+C` in an interactive console. If this would happen inside of a bigger system, the entire service might become unresponsive.

```
cputest <- function(){
  A <- matrix(rnorm(1e7), 1e3);
  B <- svd(A);
}
```

In RAppArmor we have actuall two different options to deal with this. The first one is setting the `RLIMIT_CPU` value. This will cause the kernel to kill the process after 5 seconds:

```
> eval.secure(cputest(), RLIMIT_CPU=5)
RLIMIT_CPU:
Previous limits: soft=-1; hard=-1
Current limits: soft=5; hard=5

NULL
> Sys.time()
[1] "2012-07-08 17:08:35 CEST"
>
```

However, this is actually a bit of a harsh measure: because the kernel actually terminates the process after 5 seconds we have no control over what should happen, nor can we throw an informative error. Setting `RLIMIT_CPU` is a bit like starting a job with a self-destruction timer. A more elegant solution is to terminate the process from `R` using the `timeout` argument from the `eval.secure` function. Because the actual job is processed in a fork, the parent process stays responsive, and is used to kill the child process.

```
> Sys.time()
[1] "2012-07-08 16:59:06 CEST"
> eval.secure(cputest(), timeout=5)
Error: R call did not return within 5 seconds. Terminating process.
> Sys.time()
[1] "2012-07-08 16:59:11 CEST"
```

One could even consider a Double Dutch solution by setting both `timeout` and a slightly higher value for `RLIMIT_CPU`, so that if all else fails, the kernel will end up killing the process.

B.5. Fork Bomb

A fork bomb is a process that spawns many child processes, which often results in the operating system getting stuck to a point where it has to be rebooted. Doing a fork bomb in R is quite easy and requires no special privileges:

```
forkbomb <- function(){
  repeat{
    parallel::mcparallel(forkbomb());
  }
}
```

Do not run call this function outside sandbox, because it will make the machine unresponsive. However, inside our sandbox we can use the `RLIMIT_NPROC` to limit the number of processes the user is allowed to own:

```
> eval.secure(forkbomb(), RLIMIT_NPROC = 20)
RLIMIT_NPROC:
Previous limits: soft=39048; hard=39048
Current limits: soft=20; hard=20
Error in mcfork() :
  unable to fork, possible reason: Resource temporarily unavailable
```

Note that the process count is based on the Linux user. Hence if the same Linux user already has a number of other processes, which is usually the case for non-system users, the cap has to be higher than this number. Different processes owned by a single user can enforce different `NPROC` limits, however in the actual process count all active processes from the current user are taken into account. Therefore it might make sense to create a separate Linux system user that is only used to process R jobs. That way `RLIMIT_NPROC` actually corresponds to the number of `R` processes. The `eval.secure` function has arguments `uid` and `gid` that can be used to switch Linux users before evaluating the call.

References

Abu Rajab M, Zarfoss J, Monrose F, Terzis A (2006). "A multifaceted approach to understanding the botnet phenomenon." In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 41–52. ACM.

Armbrust M, *et al.* (2010). "A view of cloud computing." *Communications of the ACM*, **53**(4), 50–58.

Banfield J (1999). "Rweb: Web-based statistical analysis." *Journal of Statistical Software*, **4**(i01).

Bates D, Eddelbuettel D (2004). *Using R on Debian: Past, Present, and Future.*

Bates D, Maechler M, Bolker B (2011). *lme4: Linear mixed-effects models using S4 classes.* R package version 0.999375-39, URL http://CRAN.R-project.org/package=lme4.

Bauer M (2006). "Paranoid penguin: an introduction to Novell AppArmor." *Linux Journal*, **2006**(148), 13.

Bertram A (2012). *Renjin: JVM-based Interpreter for R.* URL http://code.google.com/p/renjin/.

Canonical, Inc (2012). *Ubuntu 12.04 Precise Manual: GETRLIMIT(2).* URL http://manpages.ubuntu.com/manpages/precise/man2/getrlimit.2.html.

Cloudstat (2012). URL http://www.cloudstat.org/.

Dabbish L, Stuart C, Tsay J, Herbsleb J (2012). "Social coding in GitHub: transparency and collaboration in an open software repository." In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pp. 1277–1286. ACM.

Dahl D, Crawford S (2008). "Rinruby: Accessing the r interpreter from pure ruby." *Journal of Statistical Software*, **29**(4), 1–18.

Daroczi G (2012). *sandboxR: filtering "malicious" calls.* URL https://github.com/daroczig/sandboxR.

Eddelbuettel D, Francois R (2011). *RInside: C++ classes to embed R in C++ applications.* R package version 0.2.4, URL http://CRAN.R-project.org/package=RInside.

Free Software Foundation (2012). *GETRLIMIT – Linux Programmer's Manual.* URL http://www.kernel.org/doc/man-pages/online/pages/man2/setrlimit.2.html.

Harada T, Horie T, Tanaka K (2004). "Task oriented management obviates your onus on Linux." In *Linux Conference.*

Heiberger R, Neuwirth E (2009). *R through Excel: A Spreadsheet Interface for Statistics, Data Analysis, and Graphics.* Springer-Verlag New York Inc.

Horner J (2011). *rApache: Web application development with R and Apache.* URL http://www.rapache.net/.

Horner J, Eddelbuettel D (2011). *littler: a scripting front-end for GNU R*. Littler version 0.1.5, URL http://dirk.eddelbuettel.com/code/littler.html.

Mirkovic J, Reiher P (2004). "A taxonomy of DDoS attack and DDoS defense mechanisms." *ACM SIGCOMM Computer Communication Review*, **34**(2), 39–53.

Moreira W, Warnes G (2006). "RPy (R from Python)." URL http://rpy.sourceforge.net/rpy/README.

Ooms J (2010). *yeroon.net/lme4: A web interface for the R package lme4*. URL http://rweb.stat.ucla.edu/lme4.

Ooms JC (2009). *stockplot: a web interface for plotting historical stock values*. URL http://rweb.stat.ucla.edu/stockplot.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Smalley S, Vance C, Salamon W (2001). "Implementing SELinux as a Linux security module." *NAI Labs Report*, **1**, 43.

Torvalds L, Hamano J (2006). "GIT-fast version control system."

Urbanek S (2011). *Rserve: Binary R server*. R package version 0.6-5, URL http://CRAN.R-project.org/package=Rserve.

Wickham H (2009). *ggplot2: elegant graphics for data analysis*. Springer New York. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Affiliation:

Jeroen Ooms
UCLA Department of Statistics
University of California
E-mail: jeroen.ooms@stat.ucla.edu
URL: http://www.stat.ucla.edu/~jeroen

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd