February 21, 2018

Application live-upgrading and error-recovery using code-data decoupling

Till Smejkal
Hewlett Packard Enterprise

Dejan Milojicic
Hewlett Packard Enterprise

Paolo Faraboschi
Hewlett Packard Enterprise

Application live-upgrading and error-recovery using code-data decoupling

Abstract

When applications have critical bugs that present security vulnerabilities or may result in serious failures with potential massive business level impact, these applications have to be updated as fast as possible to minimize the harm of the bug. However, mission-critical or other user-facing applications may maintain critical internal state that has to be serialized and restored during the update process introducing signi1cant cost and delay.
Instead of serializing the internal state we propose to implement applications in such a way that the application state is fully decoupled (e.g. in a different address space or shared memory segment) from the application logic. Such a decoupling allows for example that upgrades can happen without serialization of the data, even allowing side-by-side execution of the updated and the failing version of the application and thereby reducing application downtime during the update process. Furthermore, this decoupling also allows applications to recover easily from failures by recovering the previous data of the crashed application instance.

Problems Solved

This disclosure solves the following problems:

- Upgrading of applications to new versions with zero downtime
- Recovery from application failures
- Upgrading of shared libraries without application restart
- Preventing of ROP attacks based on shared libraries

Prior Solutions

This disclosure is related to the following other disclosures:

- MVAS core disclosure
- Application specific and transparent checkpointing
- Upgrading applications by stopping them and restarting

Outline Solution Description

In usual programs, application logic and application state reside in the same virtual address space and are not further separated from each other. Accordingly, if an application crashes or has to be terminated in order to be updated, the application's current state is lost together with the application. We propose, to decouple the state of the application from the code part in such a way that the application state can exist independently of the process of the application. Thus allowing it to be reused if the application crashes unexpectedly or has to be updated. Possible ways to decouple the application state is to use functionalities such as the shared memory support of Linux (SHM), multiple virtual address spaces (MVAS) or memory-mapped files resulting in an address space layout as shown in Figure 1. Out of this three solutions, SHM and MVAS are the most interesting ones, because they don't introduce further performance overhead as it can occur when using memory-mapped files. However, if the operating system does not provide either SHM or MVAS, the application can fall back to

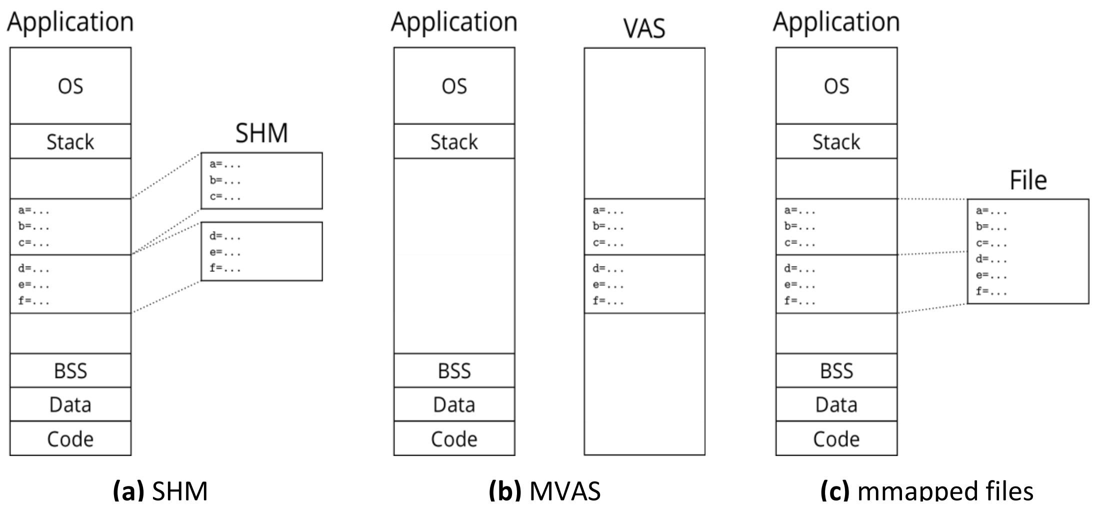


Figure 1 The application's address space layout when built with code-data decoupling based on SHM (a), MVAS (b), or a memory-mapped file (c).

using plain memory-mapped files to keep its state decoupled.

When an application uses the proposed technique to decouple its state and code memory regions, the application's overall execution would follow a scheme as outlined in Figure 2.

Before actually starting the main() function of the application, the memory region(s) where the application's state should be stored in have to be initialized. That means, depending on the technique which is used by the application, a new virtual address space has to be created, a new SHM region has to be allocated, or a new file has to be initialized and memory mapped in the address space (marked with (1) in the figure). Just after this setup phase is completed, the application can start its actual execution which must use the provided memory regions to save internal state. One possible technique to ensure that the application's state is saved in the corresponding memory regions is to install a specialized malloc() and free() function in the application together with the initialization of the memory region.

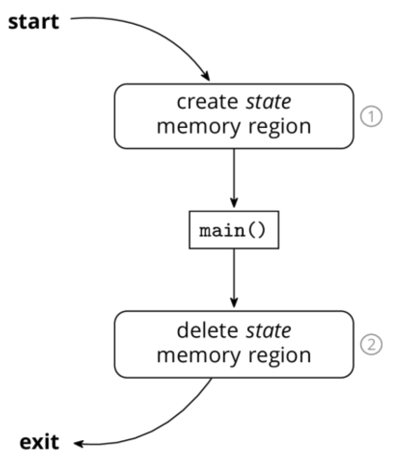
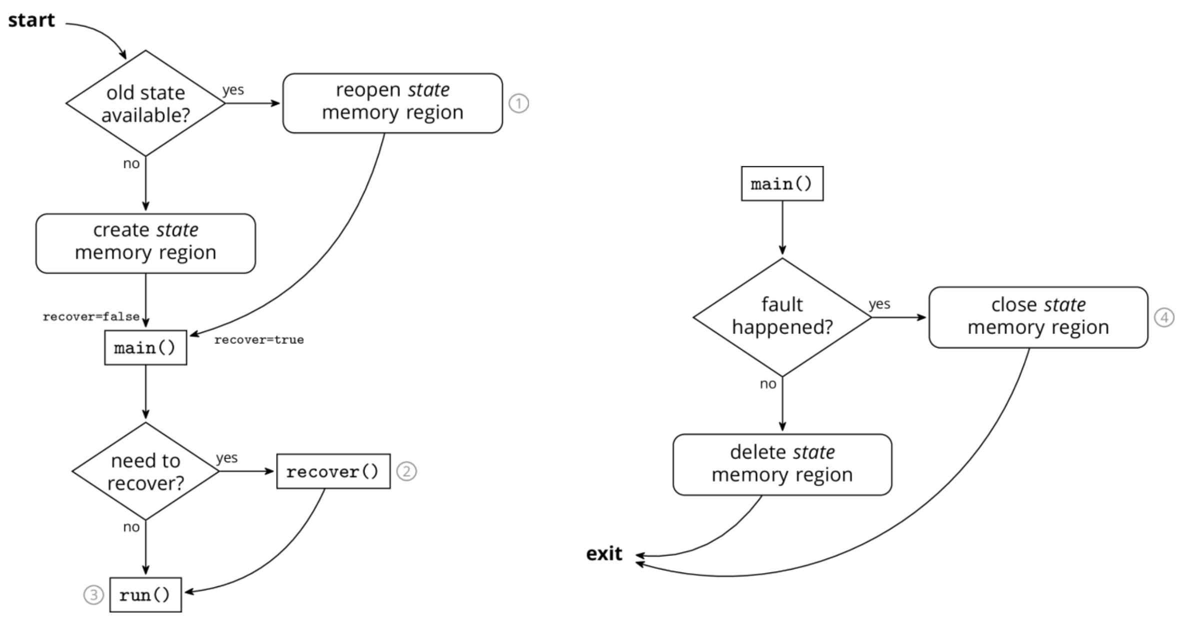


Figure 2 Execution scheme of an application using code-data decoupling.

After the application finishes its normal execution and exits from its main() function, an additional step is necessary before the application can exit completely. The created memory region that contains the application's state has to be deleted again together with the data allocated in the backing storage technique (step (2) in the figure). This step is only needed if multiple application instances are independent and cannot reuse existing data.

Recovery from Application Crashes

By combining the code-data decoupling technique with additional support in the application and some additional information that are kept together with the backing storage technique (SHM, MVAS, or memory-mapped files), recovery from application crashes can easily be added to an existing program that already uses code-data decoupling. The startup and shutdown of the application only have to be slightly adapted as outlined in Figure 3.



(a) Start up with crash recovery support. **(b)** Shutdown with crash recovery support.

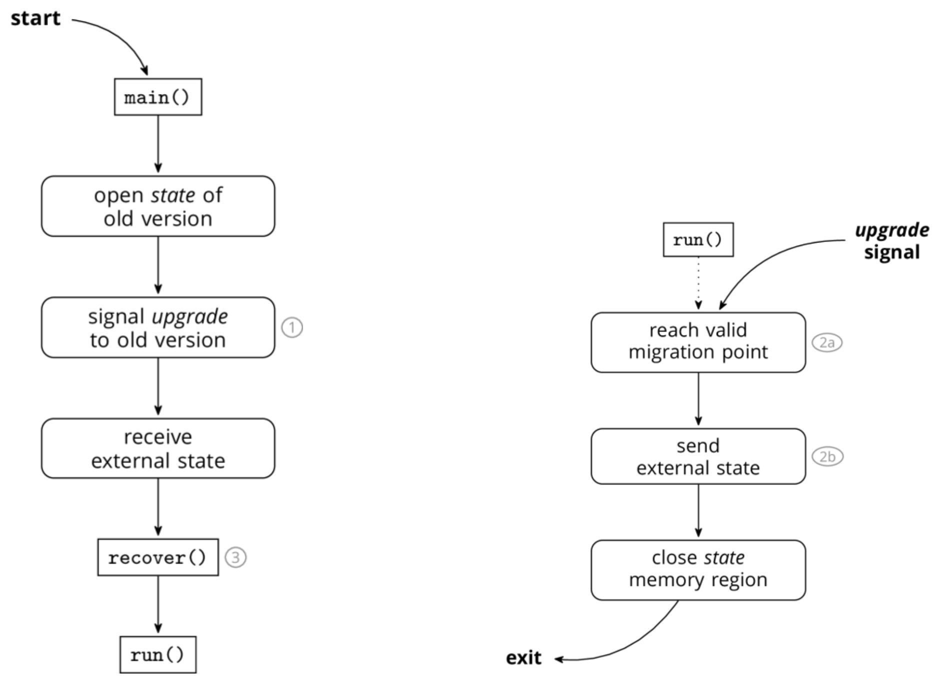
Figure 3 Execution schemes of an application that supports recovery from previous application crashes when it starts up (a) and when it shuts down again (b).

Instead of always creating a new memory region for the application state during the startup of the application, the application can check whether there is still an old version available from which the application can recover. To be able to distinguish between crashed states and states of applications that are currently running one can for example remember together with the state to which application instance it belongs (e.g. by saving the programs PID value with the state in the used backing storage technique). With this additional information, one can easily identify states that belong to crashed applications simply by checking if there is still a program running with the saved PID. If there is no old state available from which the application can recover a new state memory region together with the backing storage is created as described previously and the application executes as usually. However, if there is an old state available, it is now possible to reopen this state and reuse it in the new application (marked with (1) in the figure). Depending on the backing storage technique different steps have to be executed to achieve this functionality. After the old application state was successfully reopen, the application's start up is continued, with an indicator that a recoverable state is available. This indicator causes the application to try a recovery from the previous state (2) before it resumes its typical execution ((3) in the figure). Depending on the application and the type of fault causing the crash, some old states are not properly recoverable. If such a situation occurs, the programmer can decide on its own how to handle it. One possible reaction could be to create a new state memory region, drop the old one, and just continue with a new execution state. Alternatively, the application can also abort and report to the user, that it was not able to recover from a previous state of a crashed instance.

In order that the state recovery during the application start up can be used as described, some additional steps have to be performed during application shutdown as well. If the program exits from its normal execution routine (e.g. its main() function) it is necessary to check whether a fault happened in the application, or if this is a normal shutdown. If no fault happened, the state memory region can be deleted together with the backing storage as described before. However, if a fault happened the state must not be deleted, since it might be that the application wants to recover from the state again later. Hence, depending on the used storage technique, the memory region should only be closed or unmapped, but the backing storage must not be deleted (step (4) in the figure). Accordingly, this means that there might be no additional step necessary during the shutdown if a fault happens. For example, if memory-mapped files are used as backing storage, no additional steps are necessary in a fault, because the file will be unmapped and closed by the operating system already when the program is killed.

Live Upgrading to new Application Versions

Based on a similar approach as used for the crash recovery, one can also implement live-upgrades of applications to newer version. This technique is especially interesting for situations when one wants to deploy urgent security fixes for applications without causing any downtime of the services that they provide. Application live-upgrading based on code-data decoupling follows the scheme as outlined in Figure 4.


start
main()
open *state* of old version
signal *upgrade* to old version
1
receive external state
recover()
3
run()
run()
upgrade signal
reach valid migration point
2a
send external state
2b
close *state* memory region
exit

(**a**) Upgrade process for a new application version.

(**b**) Upgrade process for the old application version.

Figure 4 Execution schemes of an application that uses live-upgrading based on code-data decoupling whereas (a) describes the steps done by the new version and (b) the steps of the old version of the application.

Immediately after the new version of the application starts, it searches and opens the state memory region of the old version from the storage technique that is used in the background to realize the code-data decoupling. Now that it has a reference to the state of the running application, it sends a signal to the corresponding program (1). This signal causes the old program version to stop its current execution and run some special handler code. As a very first step, the signal handler code makes sure that the application is currently in a state at which upgrading the application is possible and if not let the old application version continues running until it eventually reaches a proper state (step (2a) in the figure). If the handler code is sure that upgrading is finally possible it transfers as next step all other external state (e.g. open files, network sockets, pipes with other programs, etc.) from the old version to the new version of the program (marked with a (2b) in the figure), since this part of the application state is not contained in the memory region but is crucial for proper recovery in the new application version. To be able to migrate such state, corresponding support must be available in the operating system. When the transfer of this external state is finished, the old application version can close its reference to the now shared state memory

region and can exit. The new application version on the other side, which now has all the necessary information can fully recover the whole state of the old program version (step (3) in the figure) and can afterwards continue the execution where the old version stopped.
This described way to implement live-upgrading of an application requires, that the internal data representation of the application did not change within the version update because the new application version directly reuses the in-memory data of the old program version and hence requires the exact same data structure layout. However, if the update contains a change in a data structure, there are multiple ways to still be able to perform live upgrades. One possibility is to add an additional translation phase in the recovery phase of the new version of the application that translates all old data structures into the new layout. Unfortunately, this can massively increase the time that the update requires which can be problematic in some situations. Another possible solution is to not translate the data structures in one step, but perform the translation on the fly whenever the data structure is used and then perform another live-upgrade to another version that only understands the new data structure layout once all old version were translated. This approach reduces the cost that has to be paid during the first update but adds some constant overhead and makes the application significantly more complicated.
The update process, especially the transfer of the external application state can be further simplified if the operating system provides various buffering mechanisms. So is it for example possible to not transfer DBus[1] names or listening network sockets but instead open them again in the new application version, if the kernel provides the appropriate buffering.

Decoupling of Shared Libraries and Transparent Restart

Another possible usage of code-data decoupling which is especially powerful if implemented with multiple virtual address spaces (MVAS) is to decouple shared libraries from the actual application address space. Instead of directly mapping the shared library code in the address spaces of the application, the library is kept separated from the application and only made accessible whenever the application executes a function provided by the library. A possible implementation that uses this technique with the help of MVAS is shown in Figure 5. The code and data of the shared library is managed in a dedicated virtual address space and only a small trampoline code is actually mapped in the application address space.

[1] DBus is a widely used inter-process communication technique for Linux desktop applications.

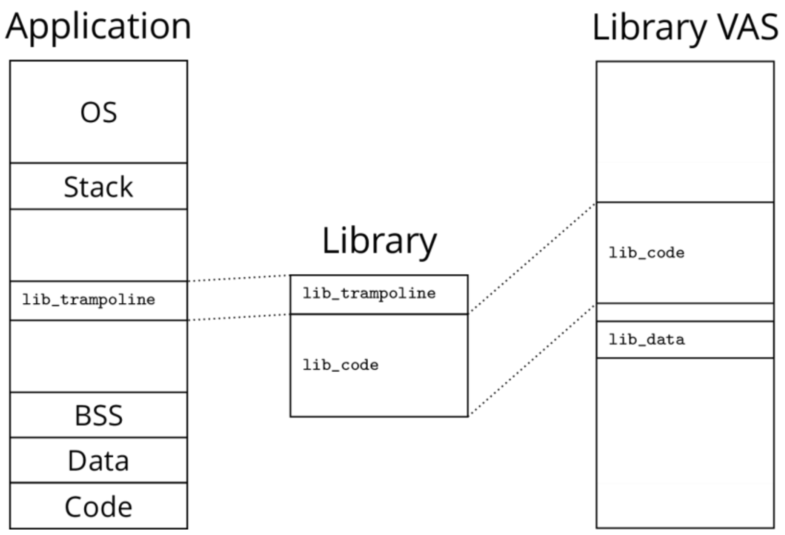


Figure 5 Address space layout of an application with decoupled shared libraries.

Whenever the application wants to use a function that is provided by the corresponding shared library, the application will not call the library code directly but instead will end up in a specialized wrapper function in the trampoline code ((1) in figure 6), that follows a simple execution scheme as outlined in Figure 6.

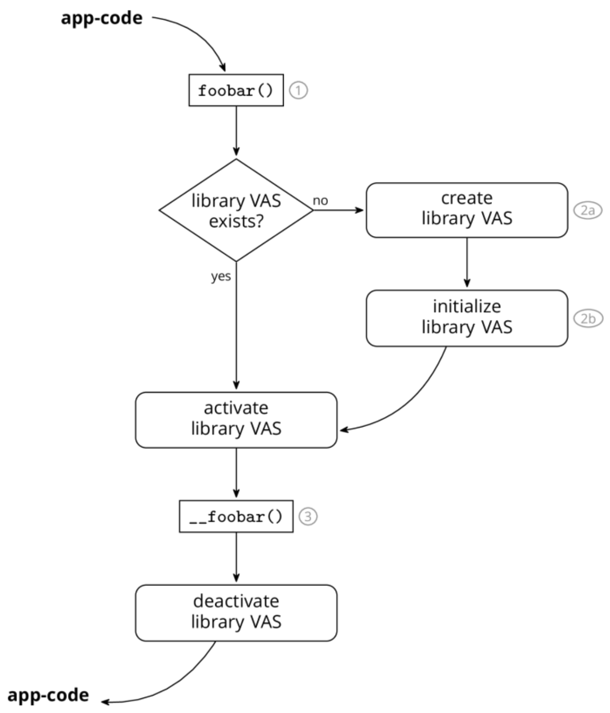


Figure 6 Execution scheme for an example function foobar() provided by a decoupled library using MVAS as storing technique.

As a first step, the wrapper function will check whether there is already the decoupled library code available for the application. If the decoupling is realized with MVAS, the trampoline code checks whether there is already a special library address space created and attached to the process. If not, the trampoline will create a new virtual address space[2] (VAS) for the library (step (2a) in the figure) and will initialize it by mapping the actual library code into it and creating an empty data memory segment where the library can save internal management data (2b). If the library address space already exists, or if it is properly initialized, the trampoline code will eventually activate the VAS and thereby make the library code accessible. Since the library code can now be accessed, the next step is to call the actual function that provides the requested functionality (marked with (3) in the figure). After this function finishes and hence before the control flow returns to the application, the VAS of the library is again deactivated by the trampoline code, thereby making the library's code and data again inaccessible from the application's address space.

If the operating system does not have support for multiple virtual address spaces, one can achieve the same behavior with SHM or simple memory-mapped 1les by mapping and unmapping the files or SHM regions in the wrapper functions in the trampoline code. However, since the approach based on MVAS does not require the mapping and unmapping, it can achieve much better performance. Especially if combined with tagging of translation look-aside buffer entries and other techniques to enhance address translation, MVAS-based library decoupling can achieve near native performance.

This decoupling of library code from the application address space has two very important advantages. On the one side, since the library code is not accessible from the application address space, there is a smaller attack vector for ROP (return-oriented-programming) attacks. The other big advantage of this decoupling is, that one can transparently for the application update the shared library to a new version. This can be achieved for example in the trampoline code. Instead of simply activating the VAS of the shared library, the trampoline code can additionally check if there is a new version of the library in the system (e.g. with an important security fix) and accordingly update the code mappings. Thereby, the shared library can be updated to a new version without any support or knowledge in the application.

In general the whole shared library decoupling technique can be implemented independently and without support of the application that uses the feature.

Advantage over Previous Solutions

The proposed disclosure contains various new techniques for application live upgrading and recovery that provide multiple advantages over existing solutions. The live-upgrading technique for example allows zero-downtime updates of applications without the need of another external decoupling service such as a load balancer or a proxy- or buffering-application. Instead the zero-downtime mechanism is directly implemented in the updated application.

The crash recovery support that comes with the proposed code-data decoupling of applications also has significant advantages over existing solutions. One advantage is for

[2] For further information about VAS and how they interact with the address space of a process, please refer to the disclosure for multiple virtual address spaces.

example, that with this technique it is not necessary to regularly create snapshots/checkpoints/backups of the application, but the backup of the application data is created implicitly because the data is kept in an independent storage. Furthermore, no serialization or deserialization of backed-up data is necessary. Instead, the backup can be used as is and if MVAS or SHM is used as backup storage technique, no access to slow disks is necessary to restore the application.
Furthermore, the shared library decoupling also has multiple advantages over existing solutions as also already mentioned in the corresponding section. With this technique, it is for example possible to reduce the attack surface for ROP-based attacks on applications. Another advantage of the shared library decoupling is the transparent library updating which allows fast and simple deployment of security fixes while still maintaining a small overhead during the executing of the application.

Authors:

Till Smejkal, Hewlett Packard Enterprise
Dejan Milojicic, Hewlett Packard Enterprise
Paolo Faraboschi, Hewlett Packard Enterprise