Dynamic Repair of Mission-Critical Applications with Runtime Snap-Ins

J. Peter Brady, Sergey Bratus, Sean Smith

Chapter 12

DYNAMIC REPAIR OF MISSION-CRITICAL APPLICATIONS WITH RUNTIME SNAP-INS

J. Peter Brady, Sergey Bratus and Sean Smith

Abstract This chapter proposes a solution that provides reliable, non-disruptive updates to critical systems using a novel design pattern called a "snap-in," which is able to install replacement routines embedded in shared libraries during system execution. Most system updates are performed in a static or maintenance state. However, dynamically updating software reduces the time required for adding functionality and applying security upgrades. The proposed snap-in solution improves on previous work by adopting the novel approach of using the target's application binary interface to first load shared libraries that contain replacement routines into a running application, supplanting the original routines with replacement routines without having to modify the existing code. An automated toolkit is provided for scanning application binaries and determining where the replacement routines are to be added.

1. Introduction

In 1992, researchers studied the software faults discovered during integration testing of the Voyager and Galileo spacecraft code at the Jet Propulsion Laboratory [17]. The bulk of the faults were directly attributed to errors in understanding or implementing requirements, and to miscommunications between development teams. Not surprisingly, "there's no such thing as a bug-free application" [32].

Not all faults in modern computing systems are found during internal integration and testing. As a result, faults found during field deployments become part of the maintenance cycle. Maintaining software during its lifetime is a significant and costly problem.

A NIST report [22] reveals that the costs to repair system defects increase rapidly after the requirements stage. Table 1, taken from the report, shows

Table 1. Repair costs during various lifecycle stages.

Lifecycle Stage	**Repair Cost**
Requirements	$1x$
System Testing	$90x$
Installation Testing	$90x$ to $440x$
Acceptance Testing	$440x$
Operation and Maintenance	$470x$ to $880x$

that the cost of repairing a defect in the field is up to 880 times the cost of repair during the requirements stage (x denotes a normalized unit of cost).

Standard application maintenance generally has the following cycle:

- **Revision:** Decide what is to be changed, such as repairs based on bug reports from users or the addition of new functionality.
- **Development:** Make the changes to the application, rebuild and test in the engineering environment, and pass a release candidate to configuration management.
- **Testing:** Run the release candidate through a quality assurance process to ensure that it is ready for release.
- **Deployment:** When the new application release is ready to be installed, shut down the old version of the application, execute an installer program that loads the new release and start the new application.

This maintenance cycle does not work well for all systems. The ability to do dynamic updating as opposed to a restart-style deployment is necessary, especially in the case of mission-critical systems that cannot have any downtime. For example, a communications satellite, the Mars Rover or a power grid cannot be switched off entirely to update their software. While the Mars Rover had extensive planning and infrastructure to allow for software updates [6], not all systems have the level of resources needed for repairs, so devising an alternative technique is imperative.

Examples of other systems that do not have the standard maintenance lifecycle are those that are obsolete or that were created by vendors who are no longer in business. An inability to update system software can have disastrous consequences, such as not being able to contain a virus like Stuxnet [13] or a potential wide-spread failure.

Industrial control systems, which operate complex and dispersed infrastructures such as electric grids, oil and gas pipelines, and power plants, are good examples of critical systems with challenging maintenance cycles. Several guides for securing industrial control systems have been published (e.g., [27]). However, concerns have been raised about hardware obsolescence [9] and that industrial control systems became operational before the latest security techniques

were developed [7]. Additionally, some industrial control systems may run on old or obsolete platforms that no longer have vendor support for their hardware and/or operating systems.

Internet of Things (IoT) devices have similar maintenance problems. Many consumer devices – as well as some industrial Internet of Things (IIoT) devices – have non-upgradeable firmware, meaning that there are no easy system upgrade paths. Additionally, integrators often incorporate low-cost circuit boards in their systems with no opportunities for firmware updates. Internet of Things software systems often have security issues. For example, developers may put together software from various sources in an *ad hoc* manner, resulting in security holes such as default or non-changeable administrative passwords [18] and buffer overflows such as those exploited by the Mirai botnet [12].

To address these challenges, this chapter presents a new design pattern called a "snap-in" that facilitates the insertion of new or modified software in a running system. Information in the application binary interface (ABI), in this case in the Linux executable linkable format (ELF), is leveraged to find faulty routines that are subsequently replaced with updated versions even while the system is operational. This novel approach ascertains information about an application and uses it to replace routines without changing the structure of the application. This is an important point because modifying application code directly can leave it in an inconsistent state.

Snap-ins are designed to quickly repair faulty code (e.g., validating SSL certificates in Internet of Things devices [15]) or to make rapid repairs to programs that experience "zero-day blooms" [24] (i.e., latent errors that can affect a wide range of programs, and program or operating system versions). Snap-ins also enable the layering of security proactively at a global control point in a piece of unmodified software [23]. For example, secure input-handling parsing of command inputs to Internet of Things devices via the application of language-theoretic security [1] can avert potential security holes by creating parser-combinators that enforce input validation to prevent malicious data manipulation.

2. Snap-in Overview

The snap-in system has three major components:

- **Shared Libraries:** Shared libraries contain the patches to be installed. The patches modify or augment the operation of the target application. Shared libraries are employed to leverage standard software engineering techniques for aggregating custom routines or for modifying later versions of shared libraries used by the target application that is being repaired.

- **Mapping Data:** This data maps system executables to the repaired routines.

- **Snap-In Controller:** The snap-in controller reads the mapping data, searches for running target executables, pauses the execution of the target

Table 2. ELF segments used by snap-ins.

Segment	Writable	Definition
`.header`	No	ELF header and segment table
`.hash`	No	Symbol hash table
`.dynsym`	No	Dynamic linking symbols
`.dynstr`	No	Dynamic linking strings
`.plt`	No	Procedure linkage table
`.text`	No	Executable code
`.rodata`	No	Read-only data
`.data`	Yes	Initialized data
`.got`	Yes	Global offset table
`.got.plt`	Yes	Global offset table for procedure linkage table
`.dynamic`	Yes	Dynamic linking information
`.bss`	Yes	Uninitialized data

application, injects the new libraries, modifies the function addresses to point to the new libraries and then resumes the target application.

Details about the operation of the snap-in toolkit are provided later in this chapter. However, in order to understand how snap-ins work, it is necessary to discuss the binary format underling Linux applications, specifically, ELF.

2.1 ELF Files

Snap-ins require the ELF ABI [28], the current standard binary executable format for Unix and Linux systems. An ELF file is a dual-use object. It initially serves as a container for a compiler to store machine code and data and for the linker to assemble all the selected files into an executable program. The compiler creates a set of sections that contain the compiled code, data, relocation information and external references to other routines. ELF establishes a section header table that is accessed by the linker to resolve and update the reference sections in each file.

When a linker creates an executable file, it writes a program header table into the resulting ELF file. The program header table points to a set of segments. A segment has zero or more sections; for example, a read-only segment may contain code in an executable text section while constants reside in a read-only section.

An ELF file is also used when a loader reads the program header table of an executable file to map the file segments into memory and resolve run-time symbols via the shared libraries. As discussed later, the file segments are also used by the snap-in program to modify the software. Table 2 lists the ELF segments used by snap-ins. Interested readers are referred to [14] for more information about ELF.

Most Linux binaries are dynamically linked – they rely on a loader to connect required external system calls or functions to the correct shared libraries. For example, if an application wishes to call function `read()`, the linker writes information into the application that says it is located in the GNC C library `libc` along with the offset in the library. The linker does not write the actual address into the application for reasons of flexibility – if one or more libraries are upgraded, then all the applications that use the libraries would have to be re-linked to work properly.

Modern Linux systems use address space layout randomization (ASLR) [26] to map the shared libraries required by an application at randomized locations in the application memory space. This approach prevents malicious applications from using memory corruption to access resources that are denied to them. Attempting to write a hard-coded address into an application would make this feature unusable.

Since a loader must ascertain the memory space at runtime and resolve the connections just before the application runs, it needs to do its job quickly. The design of the ELF file makes this possible by establishing connections to the shared library as and when they are needed.

Continuing with the example, the first time that function `read()` is called by the executable, the mainline code calls some interlude code in the procedure linkage table (PLT) that triggers the loader to connect function `read()` to `libc`. The loader places the address entry in the `.got.plt` segment following which the function `read()` is called in `libc`. Subsequent calls made to `read()` automatically use the address written in segment `.got.plt`, so only the first call to the shared library function incurs a small time penalty. This procedure is crucial to the operation of snap-ins.

2.2 Mapping Data

Since the ELF ABI underpins the target applications to be repaired as well as the shared libraries to be installed, this research has a designed a toolkit containing a program that can read either. The toolkit extracts from the ELF data the entry point name, its relocatable address and, in the case of an application, the library name pointed to by the entry. It also identifies writable data that has to be relocated.

If the target application is multi-threaded, then additional data is collected to find the best places to stop code execution safely, primarily in blocking routines such as `select()`, `sleep()`, `fork()` and `pthread_cond_wait()`. These points, which are referred to as "thread markers," are stored in the mapping data.

The collected data is then used by the snap-in controller to match the repaired routines with target applications.

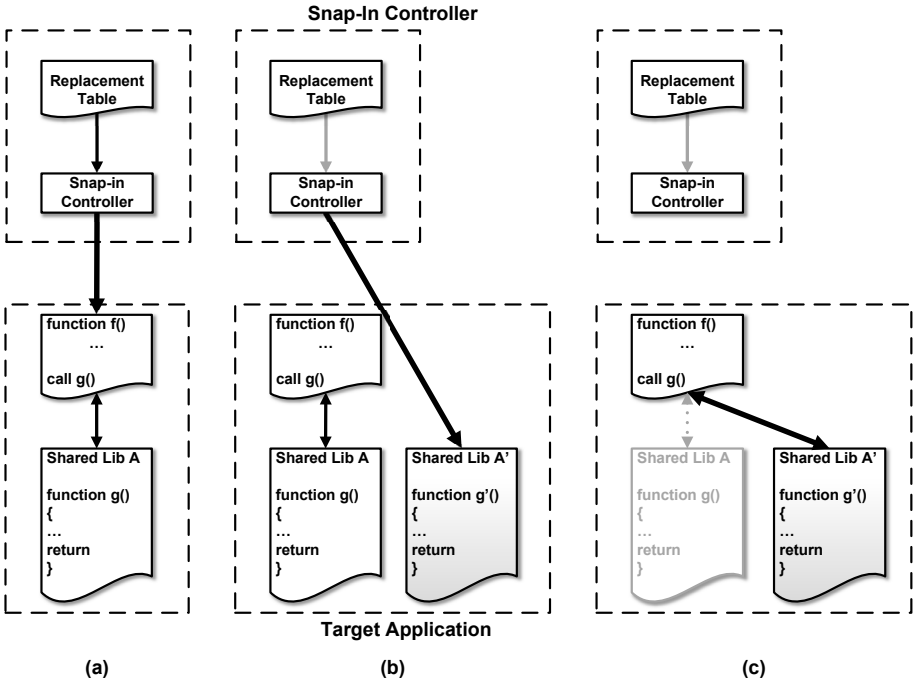

Figure 1. Snap-in controller in operation.

2.3 Snap-In Controller

The snap-in controller is a daemon that runs on the system that is to be updated. The controller uses the collected mapping data to insert shared libraries into an operator-selected set of running programs.

The controller runs in a loop to query all the running applications on the system. If an application matches the pattern in the mapping data and has not already been updated, then the controller checks to see which shared libraries need to be loaded. After the shared library information is found, the controller briefly pauses the program *in situ* in the case of single-threaded applications. In the case of multi-threaded applications, it uses thread markers in the stored data to identify the appropriate places to pause.

Figure 1 clarifies the operation of the snap-in controller. Note the controller does not perform any modifications when the target application is executing; in such a situation, the controller pauses and it tries again after a timeout period.

The controller first reads the replacement table to find the target application that is running without modifications. The running target application initially uses function `g()` in shared library `A` (Figure 1(a)). Function `g()` has a bug and will, therefore, be replaced with the repaired function `g'()` located in a new shared library `A'`. Note that if multiple shared libraries are to be installed, all of them are installed into the process memory during this step.

Next, the ELF structure is used to connect the new library calls. In Figure 1(b), the controller installs the new snap-in library `A'` into the target address space and modifies the address pointers and data to point to the new function `g'()`.

The controller first checks if the loader has completed a lazy or a full binding to the entry points in the original library. If the routine is fully bound, then the controller checks the saved program counter to ensure that `g()` is not being accessed; if it is being accessed, then the controller restarts the executable and checks the state at an alternate quiescent point or thread marker to ensure that `g()` is not running.

After the controller is sure that `g()` is not in the execution stream, the ELF segment `.got.plt` is modified to change the address of `g()` in the old shared library routine to `g'()` in the new shared library; all future calls in the executable point to `g'()`. The address also gets the proper offset to match the address space layout randomization used for the application. Interested readers are referred to [14, 29] for details about this technique.

The controller loops through all the calls to be modified using the same technique. Certain libraries, such as DIABLO [31] and ERESI [5], facilitate the rewriting of ELF binaries. However, these libraries were not employed in this work.

On the other hand, if the loader has performed a lazy binding of the library call (i.e., the dependency is loaded when referenced for the first time), the controller does an explicit binding to the new version of the call by loading its address in the `.got.plt` segment.

After the snap-in controller completes the changes, it releases its connection to the target executable, updates its internal table to mark the executable as repaired and searches for the next executable to be repaired. The target application runs with the new library (Figure 1(c)). Calls to `g()` now go to `g'()`.

3. Snap-In Toolkit

A snap-in toolkit was created as part of this research. The toolkit contains utilities for source control and for system administrators to install snap-ins on target systems.

The toolbox supports the following functions:

- **Searching Executable Targets for Patch Points:** An automated scanner reads a selectable set of executables on a target system and saves entry point and patch point data in the XML format for each executable in an executable descriptor file (EDF).

- **Creating Patches for Executable Targets:** Software engineers develop patches, which they link to shared libraries for each target executable. The automated scanner is executed on the shared libraries to create a patch control file (PCF) with XML data for each library.

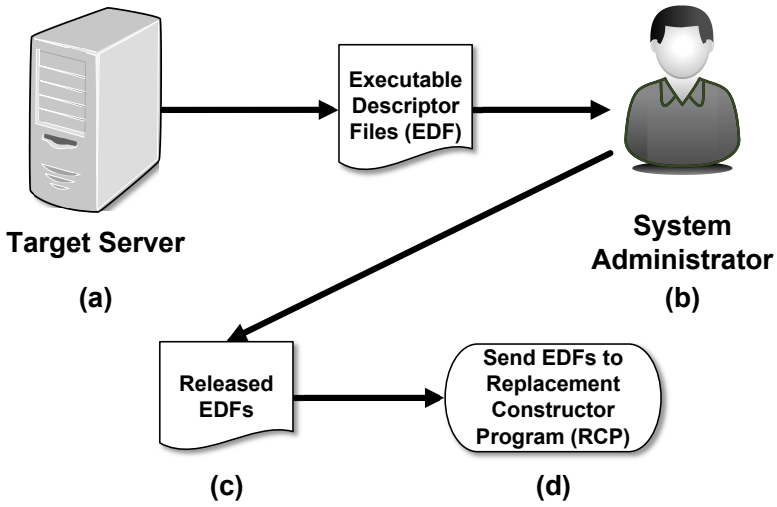

Figure 2. Creating an executable descriptor file.

- **Creating a Replacement Table:** A source code administrator obtains the patch control files for the released patches and selects the executable descriptor files associated with the target executables to be patched. A replacement constructor program (RCP) takes the two files and generates a replacement table.

- **Installing Patches on Running Executables:** A system administrator receives and loads the patches and replacement tables for each target system. The snap-in controller, which runs as a daemon on the target system, reads the replacement table and checks if the applications listed in the table are running. The patch or patches of each running application are installed automatically.

3.1 Searching Executables

A Python-based program named `snapdata` was developed to scan and read the ELF symbol table information of an executable and output the executable descriptor file. The executable descriptor file contains specifics about the executable, such as the architecture for which it is built, the system libraries and names of external entries it calls and, optionally, the location of re-entrant or threaded code.

Figure 2 shows the process of creating an executable descriptor file. The target server executes `snapdata -e` to create the executable descriptor files (Step (a)). The system administrator pulls the descriptor files (Step (b)). The system administrator decides which programs need updates and releases the descriptor files (Step (c)). Finally, the released descriptor files are sent to

```
<?xml version='1.0' encoding='utf-8'?>
<edf version="1">
<!--Executable Descriptor File (EDF)-->
<info>
<!--File location and information-->
<!--File: /usr/bin/apt-->
<path>/usr/bin</path>
<filename>apt</filename>
<class>ELFCLASS64</class>
<OS>ELFOSABI_SYSV</OS>
<type>ET_DYN</type>
<machine>EM_X86_64</machine>
<entry>0x1890</entry>
<ABI>3.2.0</ABI>
<buildID>e4e5bbe239a65880c6b7d1b9f51bfded6c61220d</buildID>
</info>
<!--External entry points-->
<entries>
<entry name="strlen"/>
<entry name="dgettext"/>
</entries>
<!--External shared libraries-->
<libraries>
<library name="libapt-private.so.0.0"/>
<library name="libapt-pkg.so.5.0"/>
<library name="libstdc++.so.6"/>
<library name="libgcc_s.so.1"/>
<library name="libc.so.6"/>
</libraries>
</edf>
```

Figure 3. Sample executable descriptor file.

another system administrator for processing with the replacement constructor program.

Figure 3 shows a sample executable descriptor file output from `/usr/bin/apt` on a Ubuntu Linux system.

Applying snap-ins while pausing all the threaded code is important to prevent state changes; therefore, it is necessary to identify locations where code execution can be stopped safely. If `snapdata` determines that an executable is threaded, it looks for natural pauses in the code – the most straightforward places are at blocking calls such as `select()`, `sleep()`, `fork()` and `pthread_cond_wait()`. These thread markers are stored by `snapdata` in the executable descriptor file. The snap-in controller uses the thread markers to pause the program when installing the replacement library.

3.2 Creating Patches

As mentioned above, patch files are standard shared libraries that contain the modified routines for a particular target library. They enable the use of a pre-built, later version of an application library as a patch, which reduces the time required to repair a critical program. For example, if a new version of an application has a bug fix and it is not possible to upgrade to this version, a library from the new version could be used without any modifications.

A shared library is just a particular type of file that contains one or more compiled object files that were built in a positionally-independent way; this enables it to be loaded into the address space of any executable when the executable is running. Building a new shared library is straightforward; interested readers are referred to [3] for details. It is important to note that the patches must line up with entry points (subroutines) for this technique to be successful.

Another way to create a patch for an old or obsolete executable with no source code is available is to translate or "lift" the target. Lifting is a process that creates an intermediate representation (IR) bytecode from the machine code of the executable. After this is done, the patch is created by modifying the intermediate representation bytecode and recompiling the fixes into a shared library for use. Lifting executables to an intermediate representation is outside the scope of this work; interested readers are referred to [8, 16, 30] for additional details.

After the shared libraries containing the patches have been created, the `snapdata` program is used to scan and read the ELF symbol table information of the executable, run with a patch scanner flag set in order to scan the libraries and create a patch control file that maps the routines in each library. The first section of the control file gives the name and version range of the target library that is modified (it can be allowed to operate on all or selected versions of the target). The second section of the descriptor file lists the routine names in the target library to be replaced. The names in the patch file should typically match those in the target library, but a command developer may map the target name to another routine name in the patch library.

Figure 4 shows the process for creating patch control files. Developers select patches and create snap-in libraries and store the completed patches on a patch server in preparation for transfer (Step (a)). A configuration manager decides when to apply the set of patches and executes `snapdata -p` on the patch server to create the patch control files (Step (b)). The configuration manager decides which patch control files are to be included in a specific release (Step (c)) and sends the released patch control files to another administrator for processing with the replacement constructor program `snaprcp`. This process enables the targeted servers to receive the new shared libraries.

The snap-in shared libraries are installed in the `/lib/snapin` directory on the target system. Keeping them in a single location is straightforward for an operator; the directory tree is protected so that only the superuser can make modifications. An operator can install all the released snap-ins on a target system, but their use by the snap-in controller is determined by the installed

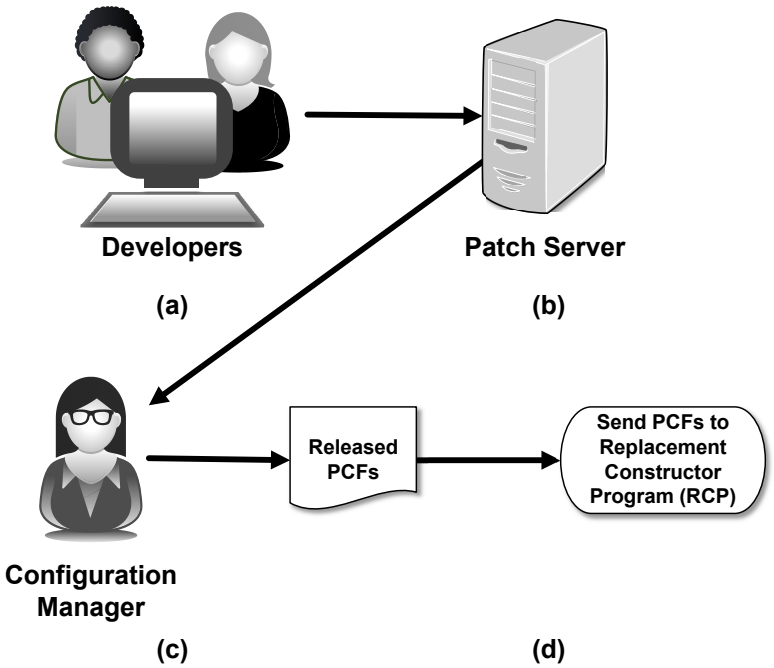

Figure 4. Creating patch control files.

replacement table, which is unique for each system. In fact, as described below, the installed replacement table may optionally be protected with encryption.

3.3 Creating a Replacement Table

The `snaprcp` tool creates a set of mappings between system executables and the created patches. The mapping data is used by the snap-in controller to decide which routines should be overridden in a program. The `snaprcp` tool reads the executable descriptor and patch control files created by parsing the executables and patches, respectively. It stores the routine of each executable and its matching patch in the replacement table for the snap-in controller.

Figure 5 shows the process involved in creating a replacement table. A system administrator receives the released executable descriptor and patch control files for a target system (Step (a)). The system administrator then executes the `snaprcp` program to produce the replacement table for the target system (Step (b)).

3.4 Installing Patches

The snap-in controller inserts new libraries into running programs and uses the replacement table to connect the appropriate subroutines to the repaired

Figure 5. Creating a replacement table.

code. The basic operation of the controller was discussed above (Figure 1), but some additional points need to be clarified.

When a shared library or libraries are installed, the snap-in controller pauses the target application to perform the installation at one of the selected thread markers and then restarts the code. When the new routines are connected via the global offset table/procedure linkage table mechanism, checks are made to ensure that the thread marker is not in the code to be changed and, in the case of a multi-threaded application, all the threads have been paused. If the conditions are not met, an alternate thread marker is chosen and the procedure is repeated until the conditions are met. If the replaced code section has static, non-constant variables, then the current states of the variables are preserved in the data section of the new routine.

3.5 Authorizing Updates

The snap-in approach enables patching without taking applications down. Mission-critical systems, such as those running in operational technology environments, require extra diligence to ensure that the snap-ins are not corrupted accidentally or maliciously. Additionally, it is important to ensure that the target system receives the correct set of snap-ins. Authorization of patches is an orthogonal question; however, the toolkit provides an option for public-key authentication [25] of updates.

Specifically, a set of unique public/private key pairs is created – one for each target machine on which the snap-in toolkit executes, one for the configuration manager to sign snap-in patches and one for the replacement table constructor. Table 3 shows how the public/private key pairs are used for signing and verification.

The following operations are available:

- **Signing Executable Descriptor Files:** The target system signs each executable descriptor file with its private key. Each target system has its own set of keys to ensure that only patches assigned to it can be loaded on the system.

Table 3. Use of public/private key pairs.

System Function	Signer	Verifier
Target Machine	Executable descriptor file	Replacement table
Configuration Manager	Patch control file	N/A
Replacement Table	Replacement table	Executable descriptor file, patch control file

- **Signing Patch Control Files and Patches:** The configuration manager uses its private key to sign each patch control file and patch that are to be delivered.

- **Signing and Encrypting a Replacement Table:** The administrator who creates a replacement table first verifies the executable descriptor and patch control files with the respective public keys. If all the files are verified, the administrator runs the replacement constructor tool as described above.

  The final operation of the replacement constructor tool is to sign the replacement table with its private key, compress the table and all the patches into a single compressed archive file, and encrypt the output with the public key of the target system.

- **Installing Snap-Ins:** When a snap-in controller detects a new compressed archive file on a target system, it attempts to decrypt the file with its private key. If this is successful, the snap-in controller attempts to install snap-ins on running applications. It then verifies the signed replacement table with its public key and continues the installation if the verification is successful.

4. Related Work

Updating software dynamically is not a new problem. Several solutions have been proposed over the past decade.

Systems such as JavAdapter [20], a runtime replacement agent for Java systems, use features of the Java Virtual Machine (JVM) along with a system of containers and proxies to replace running Java classes. While JavAdapter is platform independent, it only works with Java-based applications. In contrast, snap-ins operate on ELF binaries; they are device hardware and language agnostic and can be recompiled for any platform that uses ELF. Other formats that have defined ABIs, such as the Windows x64 ABI [19], are easily incorporated.

Ksplice [2] is an object-code layer patching system for a running Linux kernel. One or more patch files are merged with kernel source code to create a new object segment, which is loaded into kernel memory. The existing code is

modified with a trampoline to jump to the new object. However, this system only works with a Linux kernel and requires the original source code.

POLUS [4] also uses a trampoline mechanism to jump from an old function to a new one. In contrast, a snap-in modifies the pointers at the ELF level, which precludes having to modify existing code and potentially makes it easier to roll-back changes.

Kitsune [11] employs application source code and programmer-supplied transformation files to facilitate the migration of a complete process from an older to a newer version. This requires access to the original source code and the insertion of Kitsune-specific functions to control the migration. Snap-ins do not require any modifications to the original source code.

Katana [21] is the closest to the snap-in concept in that it uses ELF to do its modifications. However, it relies on source code to build patch objects whereas a snap-in does not require source code. Katana also uses a trampoline mechanism to modify the functions in running code. An advantage of Katana is cleaner migration of modified data from old to new functions; this feature will be incorporated in a future version of snap-ins.

5. Next Steps

The snap-in project is currently moving from a prototype to an initial release of the toolkit. The toolkit includes all the utilities, installation guides and sample use cases. The utilities, which are written in Python (version 2.7), are approximately 1,000 lines of code. The snap-in controller is written in C; its compiled executable is 75 KB. All releases of the toolkit will be available on GitHub (`github.com/jpbdart/snapin`).

Future versions of the snap-in toolkit will include:

- **New Algorithm for Collecting Thread Markers:** The `snapdata` collection application uses a brute-force approach to search for thread markers and data that needs to be moved. A new algorithm will be incorporated that creates a network graph of the ELF binary; this should make the algorithm faster and more accurate.

  Developers of new applications may add "quiescent points" as discussed in [10]. This would simplify the work of the `snapdata` collection application because it would only have to search for the quiescent points in code instead of looking for thread markers. The developers would be implicitly guaranteeing that the quiescent points are safe places to stop the target executables as opposed to `snapdata` making educated guesses that stopping at thread markers would not cause execution problems.

- **Rollback of Application Repairs:** The current toolbox programs collect all the data necessary to perform rollbacks. Additional code will be incorporated to enable snap-in controllers to return applications to their original running states.

- **Repairs to Statically-Linked Code:** Small Internet of Things devices and many real-time operating systems have code that is statically-linked to applications (i.e., no calls are made to external shared libraries). Efforts are underway to collect the internal program calls to facilitate code repairs.

- **Other Hardware Architectures:** The current implementation targets the Intel x86 platform. The next hardware target will be ARM. The toolbox code, which is written in Python and C, should be portable to most hardware platforms.

Another area of research is the operation of snap-ins in highly-regulated systems, such as those used in the energy sector. For example, snap-ins cannot be incorporated in a power plant control system without evaluating the changes to be made and the liability incurred in making the changes. One possibility is to obtain approval from the regulator for repairs made using snap-ins. In such a scenario, the regulator would sign off on each snap-in, adding its own authorization key to the final code along with the entity that created the code. Thus, the power plant operator would only be able to install authorized snap-ins.

Future research will also investigate the compatibility of snap-ins with real-time operating systems. As mentioned above, research is currently focusing on repairs to statically-linked applications. Once this feature is added, the toolkit collection programs should obtain the target application data that is needed. However, research has to be conducted to see how the snap-in controller can make changes to systems with hard timing constraints.

6. Conclusions

Attacks on operational technology systems, especially those that provide essential services, are increasing in scope and frequency. Even the best systems and software age from a security point-of-view, enabling attackers to discover and exploit previously-unknown holes. Quickly repairing these systems and software is of prime importance.

Snap-ins are a powerful mechanism for quickly updating system applications that cannot be shut down or that do not have traditional maintenance plans in place. Emergency repairs such as vulnerability patches and program enhancements can be seamlessly delivered in real time by snap-ins without any downtime. Security measures that prevent tampering with the patches ensure that only the correct patches are delivered to the targeted hardware.

This chapter describes work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represent that its use would not infringe privately-owned rights. Reference herein to any specific commercial product, process or service by trade name, trademark, man-

ufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation or favoring by the United States Government or any agency thereof. Additionally, the views and opinions of the authors expressed herein do not necessarily state or reflect those of the United States or any agency thereof.

Acknowledgement

This research was supported by the Office of Cybersecurity, Energy Security and Emergency Response of the U.S. Department of Energy and by the Directorate of Security Science and Technology of the U.S. Department of Homeland Security under Award No. DE-OE0000780.

References

[1] P. Anantharaman, M. Locasto, G. Ciocarlie and U. Lindqvist, Building hardened Internet-of-Things clients with language-theoretic security, *Proceedings of the IEEE Symposium on Security and Privacy Workshops*, pp. 120–126, 2017.

[2] J. Arnold and M. Kaashoek, Ksplice: Automatic rebootless kernel updates, *Proceedings of the Fourth ACM European Conference on Computer Systems*, pp. 187–198, 2009.

[3] H. Arora, Intro to Linux shared libraries (How to create shared libraries), *The Geek Stuff Blog* (`www.thegeekstuff.com/2012/06/linux-shared-libraries`), June 11, 2012.

[4] H. Chen, J. Yu, R. Chen, B. Zang and P. Yew, POLUS: A powerful live updating system, *Proceedings of the Twenty-Ninth International Conference on Software Engineering*, pp. 271–281, 2007.

[5] ERESI Team, The ERESI Reverse Engineering Software Interface (`www.eresi-project.org`), 2016.

[6] K. Finley, NASA pulls off 160-million-mile software patch, *Wired*, August 16, 2012.

[7] S. Gold, The SCADA challenge: Securing critical infrastructure, *Network Security*, vol. 2009(8), pp. 18–20, 2009.

[8] P. Goodman, Heavy lifting with McSema 2.0, *Trail of Bits Blog* (`blog.trailofbits.com/2018/01/23/heavy-lifting-with-mcsema-2-0`), January 23, 2018.

[9] H. Guzman-Miranda, L. Sterpone, M. Violante, M. Aguirre and M. Gutierrez-Rizo, Coping with the obsolescence of safety- or mission-critical embedded systems using FPGAs, *IEEE Transactions on Industrial Electronics*, vol. 58(3), pp. 814–821, 2011.

[10] C. Hayden, K. Saur, M. Hicks and J. Foster, A study of dynamic software update quiescence for multithreaded programs, *Proceedings of the Fourth International Workshop on Hot Topics in Software Upgrades*, pp. 6–10, 2012.

[11] C. Hayden, E. Smith, M. Denchev, M. Hicks and J. Foster, Kitsune: Efficient, general-purpose dynamic software updating for C, *Proceedings of the Twenty-Eighth Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 249–264, 2012.

[12] C. Kolias, G. Kambourakis, A. Stavrou and J. Voas, DDoS in the IoT: Mirai and other botnets, *IEEE Computer*, vol. 50(7), pp. 80–84, 2017.

[13] R. Langner, Stuxnet: Dissecting a cyberwarfare weapon, *IEEE Security and Privacy*, vol. 9(3), pp. 49–51, 2011.

[14] J. Levine, *Linkers and Loaders*, Morgan Kauffmann Publishers, San Francisco, California, 1999.

[15] J. Leyden, Samsung smart fridge leaves Gmail logins open to attack, *The Register*, August 24, 2015.

[16] LLVM Compiler Infrastructure, Getting Started with the LLVM System (`llvm.org/docs/GettingStarted.html`), 2019.

[17] R. Lutz, Analyzing software requirements errors in safety-critical, embedded systems, *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 126–133, 1993.

[18] D. Palmer, Is 'admin' password leaving your IoT device vulnerable to cyberattacks? *ZDNet*, April 26, 2017.

[19] M. Pietrek, Everything you need to know to start programming 64-bit Windows systems, *Microsoft Developer Network Magazine*, May 2006.

[20] M. Pukall, C. Kastner, W. Cazzola, S. Gotz, A. Grebhahn, R. Schroter and G. Saake, JavAdaptor – Flexible runtime updates of Java applications, *Software – Practice and Experience*, vol. 43(2), pp. 153–185, 2013.

[21] A. Ramaswamy, S. Bratus, S. Smith and M. Locasto, Katana: A hot patching framework for ELF executables, *Proceedings of the International Conference on Availability, Reliability and Security*, pp. 507–512, 2010.

[22] RTI International, The Economic Impacts of Inadequate Infrastructure for Software Testing, Planning Report 02-03, RTI Project No. 7007.011, Research Triangle Park, North Carolina, 2002.

[23] S. Ruoti, K. Seamons and D. Zappala, Layering security at global control points to secure unmodified software, *Proceedings of the IEEE Secure Development Conference*, pp. 42–49, 2017.

[24] S. Smith, *The Internet of Risky Things – Trusting the Devices That Surround Us*, O'Reilly Media, Sebastopol, California, 2017.

[25] S. Smith and J. Marchesini, *The Craft of System Security*, Pearson Education, Boston, Massachusetts, 2008.

[26] B. Spengler, PaX: The guaranteed end of arbitrary code execution, presented at *G-Con2*, 2003.

[27] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams and A. Hahn, Guide to Industrial Control Systems (ICS) Security, NIST Special Publication 800-82, Revision 2, National Institute of Standards and Technology, Gaithersburg, Maryland, 2015.

[28] The Santa Cruz Operation, System V Application Binary Interface, Edition 4.1, Santa Cruz, California, 1997.

[29] D. Tomaschik, GOT and PLT for pwning, *System Overlord Blog* (`systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html`), March 19, 2017.

[30] Trail of Bits, McSema, GitHub (`github.com/trailofbits/mcsema/blob/master/README.md`), 2019.

[31] L. van Put, D. Chanet, B. De Bus, B. De Sutter and K. De Bosschere, DIABLO: A reliable, retargetable and extensible link-time rewriting framework, *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, pp. 7–12, 2005.

[32] R. Varshneya, There's no such thing as a bug-free app, *Entrepreneur*, October 22, 2015.