Software security patches

Audit, deployment and hot update

Nicolas Loriant, Marc Ségura-Devillechaise, Jean-Marc Menaud

Obasco Group
École des Mines de Nantes, INRIA
4 rue Alfred Kastler
44307 Nantes, France
nloriant,msegura,jmenaud@emn.fr

ABSTRACT

Due to its ever growing complexity, software is and will probably never be 100% bug-free and secure. Therefore in most cases, software companies publish updates regularly. For the lack of time or care, or maybe because stopping an application is annoying, such updates are rarely, if ever, deployed on users' machines.

We propose an integrated tool allowing system administrators to deploy critical security updates on the fly on applications running remotely and without the intervention of the end-user. Our approach is based on Arachne, an aspect weaving system that dynamically rewrites binary code. Hence applications are still running while they are updated. Our second tool Minerve integrates Arachne within the standard updating process: Minerve takes a patch produced by `diff`, a tool that lists textual differences between two versions of a file, and eventually builds a dynamic patch that can later be woven to update the application on the fly. In addition, by translating patches into aspects and thus generating a more abstract presentation of the changes, Minerve eases auditing tasks.

1. INTRODUCTION

Despite the availability of correcting patches, in 2003, 80% of computer attacks exploited already published security vulnerabilities [3]. Sasser for example is not an exception - the patch preventing its propagation was available two weeks before it spread all over the world. Thus, most threats could be avoided by strict tracking of security bulletins and quick updating of security vulnerabilities. System administrators can not achieve these tasks without adequate tools. Indeed, reading the 5500 security alerts annually published by the CERT/CC (assuming 5 minutes per bulletin) would require about 13 weeks of work. If only one percent of the reported vulnerabilities were relevant, if the computer network is composed of one hundred machines and if updating one machine takes about an hour, deploying patches would require 157 weeks per year [2]. And this evaluation neglects the time spend in negotiations with end users to stop their applications during updates.

In this paper, we propose a semi automatic approach to deploy security updates. Its goal is to reduce the required time while still allowing system administrators to protect their network efficiently. Our framework is based on two tools, Minerve and Arachne [13]. The first reduces the time spent to audit and to adapt the patch by translating regular patches into aspect source code. The second is a dynamic weaver that deploys the translated patches on the fly freeing administrators from the hassle of negotiating with users.

This paper is organized as follows: section 2 describes a global view of our framework and shows how it integrates itself in the usual patch deployment process. Sections 3 and 4 present Minerve and Arachne respectively. Section 5 summarizes our experimental results and presents a complete example. Sections 6 and 7 discuss benefits of AOP for dynamic patching and the future work. Section 8 concludes.

2. THE FRAMEWORK

Within the open source community, security holes are corrected through the distribution of patches. A patch is produced with the `diff` tool [6], it traces the differences between the source of the old vulnerable version of the application and the source of the patched one. Hence upon a patch publication, administrators are left with no option but recompiling and redeploying the application.

Redeploying a software is very expensive with respect to time and resource consumption. First, the system administrator has to review the patch to check whether it can be trusted or not. This review is difficult as patches are not meant to be read and solely composed of the lines of source code that are different between the vulnerable version and the version of the patch. While patches stress the differences between two versions, they do not help administrators to understand the impact of the changes on the application. Secondly, patches are effective only once the application has been recompiled, redeployed and relaunched. But stopping or even suspending an application is often uncomfortable or simply impossible. Small companies running their own e-commerce site can not afford the additional costs a proper fault tolerant system forgiving temporarily unavailability of a single machine. In a roaming or mobile context, it is hard to believe that even a fault tolerant approach would ever be a solution.

The framework we propose in this paper aims at both reducing the time spent in administrative tasks and decreasing the resources required to update an application on the fly. It is worth noticing that fault tolerant approaches meet the second objective but not the first. Our approach is based on two tools: Minerve and Arachne. Minerve is a patch transformer. Its input is a `diff`-like patch. Minerve outputs a series of aspects written in the Arachne aspect language [4]. The use of an aspect language clearly presents the modifications made by the patch. Such a clear presentation decreases the time required to audit the patch. Moreover, prior to the generation of the aspect source code, Minerve checks that the new patch can be deployed dynamically without leading to an incoherent execution .

Once the system administrator has validated the new patch, our second tool, Arachne comes into action. Arachne is a run-time aspect weaver for C applications. Pre-installed on every computer of the network, Arachne dynamically weaves the patch provided by Minerve into the running program. Modifications are injected atomically ensuring the consistency of the running program.

The modifications carried out by the patch are taken into account immediately without stopping the services provided by the program or losing current work for the end-user. Nevertheless, modifications are only made on the running process, thus our framework should be complemented with a usual patching like it can be done with static patch deploying tools [8], in order for the modifications to be permanent.

3. MINERVE, PATCH ANALYSIS AND TRANSFORMATION

Source patches provided by developers are usually generated and deployed using tools like `diff` and `patch`. `diff` simply lists line by line textual differences between the two versions of every source file of a program. Thus this tool does not provide much information about the semantics of the modifications. `patch` does the opposite work by injecting differences listed into the source of the application.

In a static update process, the contents of the patch is partially validated by the compilation process. But this off-line verification does not apply to a dynamic update (on-line). From the original source code, Minerve is in charge of retrieving information about the modifications contained in the patch. This additional information permits to verify the dynamic applicability of the patch, and to produce an expressive dynamic patch that can be validated by the system administrator.

In the rest of this section, we will present how Minerve extracts, transforms and validates a patch according to the original source code of a program. In order to demonstrate the feasibility of our approach, we reviewed security holes affecting ANSI C applications running under the GNU Linux operating system on an IA32 platform. It is also important to note that Minerve does not verify the static correctness of a patch but only validates its dynamic applicability.

3.1 Modification analysis

Minerve's first task is to classify modifications contained in a patch. As we focus on applications written in C, we enumerated all possible modifications that could be applied at run-time. C is a typed, procedural language with side effects. We distinguish two kinds of types: simple types corresponding to entities that can be manipulated efficiently by the processor, *e.g. int*, and complex types made up of other types. A source patch can modify a program behavior in two ways. First, by modifying the mechanisms it contains (functions). Second, by changing type definitions of the data it manipulates (variables). From a static point of view, this distinction is unnecessary but it is essential to update application on the fly since compiled code of function bodies is usually kept in read only memory while data is not.

3.1.1 Possible modifications

Three kinds of modifications can occur: a patch can add, remove, or replace a function. Minerve treats the addition and the removal of a function as if they were function replacements. Indeed, adding a function in a running application is useless if the patch does not add another function that use it.

Replacing a function f by another function f' can possibly modify the prototype of f. This case can be seen as the addition of a new function f' while modifying all calls to f. Furthermore, when a patch replaces an existing function without changing its signature, the updating process has to guarantee that the original function is not executing at the time it is updated [7, 12]. In order to ensure this condition, we rely on Arachne' mechanisms presented in Section 4.

In order to ensure the coherency of the program, the replacement of a function f by f' must be done atomically. For this we rely on Arachne. Nevertheless this is not sufficient. Indeed to ensure coherency of the application when replacing a function, the new version f' should not read data written directly or indirectly by the execution of f because in certain cases this could lead to an incoherency. In order to ensure this, we chose to examine statically the new function f' to determine if it might use data produced by f. We use an ad hoc source code parser to check that property.

3.1.2 Modifying data's type definition

In this section we distinguish modifications made on basic types from the ones made on complex types.

Two operations have to be executed on a simple type redefinition of a variable. First, updating the value hold by the variable. Second, modifying the code that manipulates the variable. When increasing the capacity of a variable without changing its numerical type (eg: *short* → *long int*), no conversion problem can occur as the new type can always hold the current value. However, diminishing the capacity of a variable (eg: *long int* → *short*), or modifying the numerical type of the variable (eg: *int* → *float*), can only be done if the current value can be contained in the new type definition or if a conversion formula is provided. If it is possible to transfer the current value of the variable, the code manipulating it must be updated too. Indeed assembly opcodes, registers, processor flags and exceptions triggered may vary according to the size and type of operands to be manipulated. Consequently, this modification may affect surrounding instructions. In order to handle this situation generally, our tool recompiles the entire function being modified. In certain cases this might not be sufficient. Indeed, as specified in the System V Application Binary Interface [14], the responsibility of saving floating point registers belongs to the calling function, and thus modifying a variable from type *int* to *float* requires modifying the code of the calling function. Nevertheless, this case is handled by Arachne and thus modifications are always limited to the function that accesses the modified variable.

A program's behavior can also be altered by modifying a variable of a complex type definition. In this paper, we only present the addition of a new field in a structured type as it is relevant to modifications that can be made (addition, deletion, replacement). At the processor level, alteration of a structured type can modify alignment constraints on variables of that type. Some assembly instructions can have a different behavior and even not work at all when the operand they manipulate does not respect these constraints [9]. Thus, our updating process does not modify the base program code which continues to manipulate the original definition. Only the code added is aware of the new field and thus is translated to access it via a hash table indexed with the original variable's address. This solution allows us to ensure coherency of the base program without stopping it, nor needing to update all the variable at once.

3.2 Patch auditing

There are two reasons for auditing patches: to ensure that the vulnerability is really corrected, and to check that the code added by the patch does not include a new vulnerability. It can also be necessary to adapt the patch to a specific security policy. As an example, many specialists advise inserting an alarm associated with an Intrusion Detection System (IDS) in addition to the patch, in order to detect exploitation attempts [11]. The use of Arachne's aspect language make it easy for the system administrator to the add code triggering the IDS inside the patch.

Contrary to `diff` that gives very little information on the modifications contained in a patch and that presents them in a very low-level line-by-line manner, Minerve translates the patch into Arachne's aspect language. This more abstract representation of the modifications lists all functions, variables and type definitions that have been altered and their respective new version and thus eases the comprehension.

Arachne's aspect language offers an efficient join point model and high level constructs that allow to easily benefit from aspect-oriented programming [4]. Nevertheless, the dynamic patching of security violations does not make full usage of the higher level constructs.

4. ARACHNE, DYNAMIC PATCH INJECTION

In this Section, we present tools provided by Arachne that allow compiling and injecting patches into a running application.

4.1 Compilation and deployment

Arachne provides an aspect compiler and a run-time weaver. The aspect compiler, `acc`, transforms aspect source code into a native shared library. The run-time weaver, `weave`, injects this library inside the application. In addition to the verifications that are made by Minerve, `acc` ensures the dynamic patch is syntactically correct. At injection time, weave checks that references made to the application by the patch exist, partially ensuring that the patch corresponds to the right application version. Even if the patch comprises multiples aspects or rewriting points, the rewriting strategies of Arachne ensure the coherency of the application during the injection. Moreover, Arachne guarantes that on failure of the weaving process, the application remains unchanged.

4.2 Arachne inside

Arachne's weaver is used via the `weave` command, it rewrites application binary code at run-time in order to inject the aspects. This section focus on the mechanisms provided by Arachne, used by Minerve. A complete description of all of Arachne's mechanisms is available in previous publication [13]. On a Pentium processor a function call is translated into binary code as a single instruction, `call`, with an address as operand. Arachne disassembles binary code in order to find `call`*s*. To associate a function name with an address, Arachne parses the application symbol table that has been produced by the C compiler. At weaving time, Arachne loads the aspect library in to the memory of the application and rewrites previously found `call`*s* to redirect the control path to the appropriate functions in the library. A similar technique is used to rewrite accesses to variables in the heap.

Some considerations are problematic during the process we just described. Indeed, the process must guarantee the coherency of the application during the weaving. Basically, no added code should be executed before every aspect is fully woven into the program. Moreover Arachne must overcome memory isolation mechanisms and consider performance issues. Arachne solves the coherency issue by the use of locks and dynamically generated hooks that save and restore the program state. To circumvent the memory isolation, Arachne uses debugging support to insert itself inside the process's memory space.

5. EVALUATION

In this section, we have evaluate of our framework. We have applied our framework to all security advisories concerning open source C softwares published by the CERT since 2002. After a brief presentation of the CERT, we present our results over the whole test suite, and one complete example.

5.1 Test suite

CERT stands for "Computer Emergency Response Team / Coordination Center". It was created in November 1988 after the appearance of the Morris worm. It aims at training and warning about internet computing security. Its age and its independence from software editors make the CERT an international reference in security. Since 1988, it has collected an accurate database of vulnerabilities reported in softwares. We made our evaluation over all major vulnerabilities (CERT Advisories) reported in open source software since 2002. This period counts a total of 67 advisories. 30% of these concern Microsoft products, 20% other proprietary products, 10% concern embedded softwares and finally 40% open source softwares. In these last 14 advisories, we neglected 2 because they were affecting unavailable versions of the software.

In the considered vulnerabilities, about half of them are buffer overflows, 20% are format string bugs, 10% are double free bugs, 5% are integer overflows, and finally the remaining ones are combinations of the 4 previously cited. All these bugs are mainly based on assertions made by the developers on the input that are not verified at execution. Thus these vulnerabilities can be easily corrected by adding tests on input data. We verified this when auditing the patches provided by the developers for these security advisories. Indeed 90% of the patches contain modifications of function code without changing prototypes and only 10% modify type definitions.

Our experiments show that our approach can be applied successfully to all the security advisories considered.

5.2 Example

In this section, we present a full example from our test suite. The vulnerability it concerns was published in June 2002 under the reference `CA-2002-18` by the CERT. The software affected is the communication server openSSH. An integer overflow might be exploited in authentication functions of the SSH2 protocol in versions from 2.3.1p1 to 3.3. It might allow the execution of arbitrary code on the targeted host.

5.2.1 The source patch

The source patch provided by the openSSH development team modifies two functions of `sshd`: `input_userauth_info_response` and `input_userauth_info_response_pam`. The modifications only add tests on the parameter `nresp`. When the parameter is invalid, the patch calls the function `fatal` to terminate the program. As shown in listing 1, the patch does not offer much information about the semantic of the modification and useful information can only be obtained by looking at the program source code.

5.2.2 The dynamic patch

Minerve transforms the source patch into a dynamic patch that essentially contains a collection of aspects that are meant to replace vulnerable functions by their safe version. Minerve names the new and safe functions by adding the suffix "`_new`" to their original name. As a function can be called in `sshd` (as for any application) via a direct call or via the use of a function pointer, it is necessary to produce two aspects in order to replace any kind of call to the replaced function. The listing 2 shows this two aspects for the original function `input_userauth_info_response`.

The pointcut of the aspect `ReplaceFunctionCall` traps every call made with a constant address to the old function (line 2). The advice call the new version with the same parameters (line 3). Thus this aspect replaces every direct call to the function with the call to its safe version. In a similar way the second aspect, `Replacepointer` (line 5) traps every read access to the address of `input_userauth_info_response` (line 6) and returns in place of it the address of the safe version (line 7). Thus any future indirect call to `input_userauth_info_response` will be replaced by its new version.

As shown in listing 2, patches produced by Minerve ease the audit by describing modifications of the application in a language close to C. Our experiments show that the framework offers a significant reduction of the time spend to deploy patches. Indeed, excluding network transfer time, Arachne updates an application in less than $250\mu s$. And because updates are made in parallel on the entire network, the time for applying the update is independent of the network size.

6. DISCUSSION AND RELATED WORK

We intended this work in order to evaluate whether Aspect Oriented Programming in suitable for dynamic patching. It is legitimate to wonder what is the benefit of AOP in this field. We already pointed that during our experiments, Minerve did not make full use of Arachne's aspect language constructs. There are two reasons for this. First, because it is a complex task to analyze an application source code in order to infer high level rules about the modifications made by patches. Second, for most part, security patches are written in emergency. Then modifications are often limited to a single test where the vulnerability might appear, thus making patches less crosscutting. Nevertheless, our experiment on larger patches show an interesting potential for AOP.

To our knowledge, no other work provides both coherency analysis and dynamic updating. Previous work has focused on determining when in the execution flow an update may be applied safely [7], without the ability to guarantee that such a moment is reachable. In contrast to this, our approach tries to determine the applicability of a patch independently of the execution.

Dynamic patching also benefits from AOP because aspects are far more comprehensive than patches, indeed, reasoning about the program execution is easier than on its code. Also, AOP is more appropriate for dynamic patching than binary rewriting APIs like Dyninst [1] or Vulcan [5]. First, aspect code is far more intuitive to read than a program. Second, when using binary rewriting APIs, the developer is responsible to ensure the program won't behave abnormally whereas our dynamic weaver, Arachne, ensures that modifications are made atomically.

7. FUTURE WORK

Our analysis of dynamic applicability of patches are for now limited to simple cases, mainly due to source code parsing. It is necessary to base our analysis on higher level representation of source code in order to avoid this limitation. Thus, we plan to use the type propagation tool Lackwit [10].

For technical reasons our framework is limited to open source C applications running on Linux, IA-32 architecture. Nevertheless, as compiled aspects are independent of the woven program, there is no restrictions for software editors to diffuse Arachne compiled patches to be applied on binary distributed programs. This would permit dynamic patching without needing the application's source code. We also plan to adapt our framework to integrate other languages and platforms and to apply it to kernel code in the near future.

8. CONCLUSION

In this paper we have presented a novel approach for security updates based on a framework for dynamic software updates. Our first tool Minerve determines whether a patch can be deployed on the fly. The use of an aspect language

```
    authctxt->postponed = 0;      /* reset */
    nresp = packet_get_int();
+   if (nresp > 100)
+       fatal("input_userauth_info_response:␣nresp␣too␣big␣%u", nresp);
    if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
```

Listing 1: The source patch correcting the vulnerability `CA-2002-18`

```
ReplaceFunctionCall :
    call(void input_userauth_info_response(int, u_int32_t, void*)) && args(type, seq, ctxt)
         then input_userauth_info_response_new(type, seq, ctxt);

ReplacePointer :
    readGlobal(void* (input_userauth_info_response)(int, u_int32_t, void*))
        then return &input_userauth_info_response_new;
```

Listing 2: The aspect patch correcting the vulnerability `CA-2002-18`

allows administrators to validate more rapidly patches. Our second tool, Arachne applies patches dynamically without data loses and makes security updates effective immediately. Moreover our framework can easily be integrated in the static update process as it accepts standard patches published by software developers.

Despite the potential existence of patches that might not be translated in dynamically injectable aspects, our framework is efficient enough to be applied successfully on all the security advisories published by the CERT since 2002.

9. ACKNOWLEDGEMENTS

This work is supported by a regional grant from the Pays de la Loire, France. The authors would like to thank Thomas Fritz for his valuable comments.

10. REFERENCES

[1] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[2] CERT/CC. Incident and vulnerability trends, May 2003. `http://www.cert.org/present/cert-overview-trends/`.

[3] Devoteam. European study on computer network security. Technical report, XP Conseil, 2004. `http://www.devoteam.com`.

[4] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system-level applications with Arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, Mar. 2005. to appear.

[5] A. Edwards, A. Srivastava, and H. Vo. Vulcan. Technical Report MSR-TR-99-76, Microsoft Research (MSR), Jan. 2001.

[6] GNU Project. Diffutils. `http://www.gnu.org/software/diffutils/diffutils.html`.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[8] HP. Novadigm: Software management, automated change management, 2004. `http://www2.novadigm.com/products/patchmanager.asp`.

[9] Intel. *IA-32 Intel Architecture Software Developer Manual*, 2001. Instruction Set Reference Manual. `ftp://download.intel.com/design/Pentium4/manuals/25366613.pdf`.

[10] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 338–348. ACM Press, 1997.

[11] B. Schneier. *Secrets and Lies : Digital Security in a Networked World*. Wiley, Aug. 2000.

[12] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2), Mar. 1993.

[13] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, Boston, Massachusetts, USA, Mar. 2003. ACM Press.

[14] U. S. L. System Unix. *System Application Binary Interface and Intel 386 Architecture Processor Supplement.* Prentice Hall Trade, 1994.