Pre-Patched Software *

Jianing Guo
jianingguo@gmail.com

Jun Yuan
yjwhust@gmail.com

Rob Johnson
rob@cs.sunysb.edu

Stony Brook University

Abstract

Developing and deploying software patches is currently slow and labor-intensive. After software vendors discover a security bug in their product, they must write a patch, test it thoroughly, and distribute it to users, who may peform further testing before installing the patch. These manual steps take time, leaving users vulnerable for days or even weeks after a bug is discovered. Pre-patched software removes these time-consuming steps from the vulnerability-response critical path, reducing the window of vulnerability to hours or even minutes. Pre-patched applications ship with latent run-time checks that are automatically inserted during the compilation process. The compiler emits checks to cover any potentially-unsafe operation in the code. When the software vendor discovers a new vulnerability in its product, it can issue an alert informing its customers that they should activate one or more of the checks. Generating the run-time checks in advance removes the manual patch-development and testing processes from the vulnerability response critical path. Thus, when the vendor discovers a new vulnerability, it can immediately issue an alert and users can act on that alert without hesitation. By default, the run-time checks are disabled and hence incur little or no overhead. We have developed a CIL-based program-transformation that pre-patches C programs for memory-safety bugs. Early experiments suggest that pre-patched software may incur little measurable run-time overhead.

1 Introduction

Software makers currently have two imperfect responses to security bugs in deployed code: run-time checks and patches. Run-time checks, such as those used by CCured[3] or Java, prevent attackers from exploiting security bugs but can have high run-time costs. Because of the performance costs, most applications ship without run-time checks. Patches, on the other hand, are created after a defect is discovered, so they have no run-time cost, but creating patches is time-consuming and error-prone. Furthermore, many users, such as server administrators, test patches before installing them because patches may contain new bugs. This leaves a large window-of-vulnerability between the time a bug is discovered and the time that users are protected.

This paper describes pre-patched software, a new technique that combines the advantages of patching and run-time checks for dealing with defects in deployed systems. Pre-patched programs ship with a set of latent run-time checks generated at compile-time and embedded in their code. A program can subsequently be "patched" by activating one or more of the latent checks. Until a check is activated, it will incur little or no run-time overhead.

The primary benefit of this approach is to move patch development and testing out of the critical path for responding to a newly discovered vulnerability. Since the run-time checks are generated at compile time and shipped with the original software, the vendor and user can test the patches in advance. When the vendor discovers a new vulnerability, it only needs to issue an alert informing its users to activate one of the latent checks. The users can act on the alert immediately and without hesitation. If users configure their computers to automatically respond to alerts, then only a few minutes may elapse between vulnerability discovery and patch installation.

2 Related Work

Numerous researchers have developed program transformations that enforce security properties at run-time: CCured[3] enforces type-safety, CRED[7] enforces memory safety, dynamic taint-tracking[9] pre-

---

* This research has been funded by NSF grants CT-0627645 and IIS-0713211.

vents input validation exploits, RICH[1] catches integer overflows. These transformations can add significant run-time overhead; transformed programs may run 1.5 to 10 times slower.

Other researchers have developed methods for automatically generating network filters for observed attacks[6, 5, 8]. All such systems face the challenge of constructing as precise and general a filter as possible from only a few observed attacks: too narrow a filter may miss future variants of the attack, too broad will reject valid traffic. This has led to the notions of attack-based detectors, i.e. filters specific to a particular worm or attack, and vulnerability-based detectors, i.e. filters that can detect any worm that targets a particular vulnerability. Obviously, vulnerability-based filters are preferred. Brumley, et al. produced the first vulnerability-based filter-generator by slicing the vulnerable application to construct a pared-down program that recognizes inputs that excercise the vulnerable application execution-path[2]. Our system also provides vulnerability-based defense but, since the patches are generated in advance, offers faster response times and increased reliability.

Wang, et al's Shield project uses network filters to prevent attackers from exploiting known vulnerabilities[8]. Shield was specifically designed in response to the unreliability and irreversibility of software patching. Pre-patched software also addresses these issues, but goes even further. With pre-patched software, patches are reliable, reversible, and *predictable*. Pre-patched software offers other advantages, as well. Shield filters must be generated after a vulnerability is discovered and often duplicate a substantial amount of the vulnerable application's logic. Pre-patched software will generate patches in advance and will not duplicate application logic, offering faster response times and potentially lower overhead.

3 Pre-Patched Software

Figure 1 presents the major differences between pre-patched software and conventional software patches. The traditional software distribution model makes no preparations for handling vulnerabilities discovered in deployed software. After the vendor discovers a vulnerability, it must develop a patch from scratch, test it thoroughly, and distribute the patch to its customers. Cautious customers may then conduct their own testing before finally installing the patch on production machines. These manual steps may take weeks or even months, during which time customers are vulnerable to attack. This is especially problematic when hackers discover the vulnerability before the vendor does.

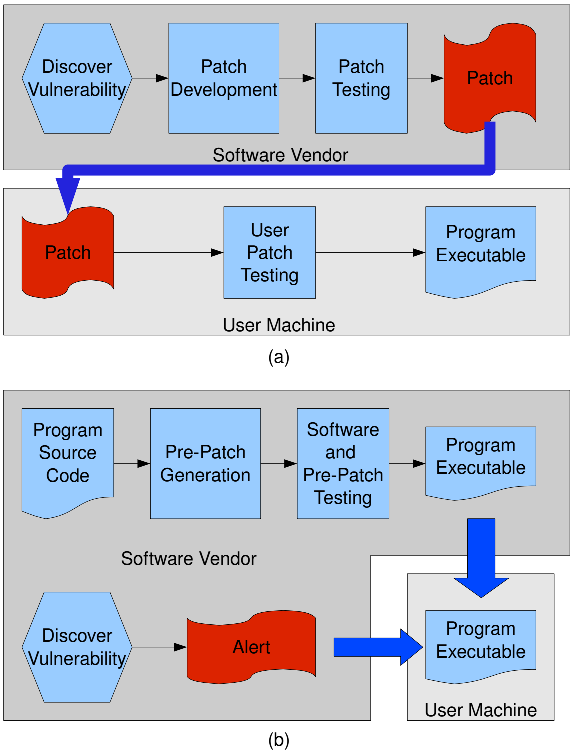

Figure 1: (a) The steps of traditional patch development and deployment. (b) Software development and "patch" deployment with pre-patched software.

Pre-patched software removes these steps from the critical path for responding to a newly-discovered vulnerability. A pre-patching compiler generates patches at compile time for potential bugs that may be discovered in the future. The vendor can test these patches before shipping the software. The patches are shipped to customers with the software. Customers can perform further testing on the patches to ensure that the patches will not break functionality used at the customer's site. Once customers are satisfied that the patches are safe, they can configure their computers to automatically activate patches upon receiving an alert from the software vendor.

Generating patches at compile time. Several research projects have produced program transformations that instrument programs to catch security bugs at run time, including memory errors[4, 3, 10, 7], SQL injection attacks[9], and integer overflows[1]. A pre-patching compiler can use similar techniques, but must leave the inserted instrumentation latent until it is needed. The compiler can generate latent checks by guarding each instrumentation site with a branch conditioned on the value of some global variable. Each instrumentation site can have its own guard variable, enabling individual instrumentations to be activated separately. Alternatively, the compiler can generate the instrumentations, copy them to an unused portion of the executable, and replace them with NOPs in the original program. Instrumentations could then be activated by copying their code back into place in the main body of the code.

In either case, the overhead of checking guard variables or executing NOPs may be non-trivial. Since almost all the instrumentation will be disabled at any given time, we can optimize this common case by generating fast-path and slow-path versions of each function. If the function has no active instrumentation, then it will execute the fast-path version, which will contain no instrumentation or NOPs and hence will run at full speed. If the function contains at least one active instrumentation site, then it will execute the slow-path version, which will contain full instrumentation.

Testing pre-patched software. Although pre-patched software can enable patches individually, vendors need only test their software with all the run-time checks enabled. Testing in this configuration may reveal latent bugs in their application, such as benign buffer overflows, that should be corrected before releasing the software. With a correct pre-patching compiler, the vendor does not need to test the software with some or all of the checks disabled. A correct pre-patching compiler should guarantee that any application that runs with all checks active will also run with any other set of active checks.

Alert generation. The software vendor can use pre-patching to respond quickly to zero-day worms and other exploits against their application. Upon discovering such an exploit, the vendor must generate an alert informing its customers to activate a check in the software. To determine which check must be activated, the vendor can simply run the exploit against an instance of the application with all instrumentation activated. The software will abort on some failing check. The vendor then issues an alert for that check. Thus, in the presence of a zero-day worm, the vendor can discover the relevant patch and issue an alert very quickly, perhaps in just a few minutes.

Limits of pre-patched software. Since the run-time checks must be generated automatically in advance, pre-patched software is not applicable to all kinds of security bug. For example, it would be difficult to create a program transformation to defend against application-level logic errors. Pre-patched software is most applicable to low-level security bugs that many applications must avoid, e.g. buffer overflows, format-string bugs, SQL injection bugs, cross-site-scripting bugs, etc. Also, pre-patched software is only as good as the pre-patching compiler. If the vendor discovers a bug not handled by the pre-patching compiler, then it must respond with standard patches.

Pre-patched software is not intended as the final response to discovered vulnerabilities. Developers can almost always write a better patch, with lower overhead and better error response, than an automated compiler. Pre-patched software prevents worms and other exploits while the vendor manually crafts a response to the vulnerability.

4 A C Memory-Safety Pre-Patcher

This section describes a prototype pre-patching transformation we are developing to protect against buffer overflows in C. Our transformation follows the general approach of the Jones and Kelly[4] bounds checker, although we have made several enhancements described below. We chose to start with the Jones and Kelly bounds checker, as opposed to CCured[3] or the memory-safe C compiler[10], because Jones and Kelly's approach requires no changes to the input source code and has a high degree of compatibility between transformed and un-transformed code. The Jones and Kelly bounds-checker's meta-data also has a simple organization, making it easier to activate individual instrumentation sites.

The Jones and Kelly Bounds Checker. A bounds-checking transformation for C must track the bounds for pointers in the program and instrument certain pointer operations to ensure that no out-of-bounds pointer dereference occurs. The Jones and Kelly transformation stores all bounds information in an interval tree indexed by pointer values. As long as all pointers remain in their correct region, then lookups in this tree will return the correct bounds for any given pointer.

The Jones and Kelly bounds-checker instruments memory allocation and deallocation to register the newly allocated regions in the interval tree. Pointer arithmetic operations are annotated to verify that the result of the arithmetic points to the same region as the original pointer. Pointer dereferences are instrumented to ensure that the entire area read or written lies within the pointer's target region. Pointer assignments, including argument passing, require no instrumentation. Casts from pointer types to integer types require no instrumentation. Casts from integer types to pointer types and casts from one pointer type to another are un-checked.

Unfortunately, many C programs perform pointer arithmetic that yields intermediate results outside the original pointer's region, although the final result is within the region. Jones and Kelly do not support such programs. The CRED[7] bounds-checker extended the Jones and Kelly bounds-checker to support such out-of-bounds pointers. With CRED, whenever a pointer goes out of bounds, its value is changed to point to a data structure that holds its true value and bounds. All other C pointer operations are modified to handle these OOB pointers.

Memsafe. Our transformation, which we call Memsafe, follows the Jones and Kelly approach with a few modifications. For each local pointer variable that does not have its address taken (which we call "solid" pointers), Memsafe creates a corresponding bounds variable that holds the bounds for that pointer. The bounds variable is updated whenever the pointer is. We cannot perform a similar optimization on non-solid pointers because they may change at any time due to aliasing and multi-threading. For solid pointers, though, this optimization eliminates many lookups in the interval tree, which are the source of most overhead in the Jones and Kelly bounds checker. Whenever a solid pointer is assigned to a non-solid pointer, e.g. a global variable, we first check that the solid pointer is in bounds. This guarantees that all non-solid pointers are in-bounds and hence we can look up their bounds in the interval tree. Whenever a non-solid pointer is assigned to a solid pointer, we look up the non-solid pointer's bounds and store them in the solid pointer's bounds variable.

We also modify functions to accept bounds parameters corresponding to each pointer parameter. For backwards compatibility, we generate a wrapper function that expects no bounds parameters. This function looks up any missing bounds and calls the real function. This enables transformed code to inter-operate with untransformed code.

Storing bounds information for solid pointers reduces run-time overhead by eliminating many lookups and also enables our transformation to handle temporarily-out-of-bounds pointers in many cases. Source programs can generate and manipulate out-of-bounds pointers as long as those pointers are not dereferenced and are stored in solid variables. Experimental results in the next section show that many programs satisfy this requirement. This enables our transformation to support most programs without the complexity of CRED-style OOB structures.

Our current prototype performs several peephole and loop optimizations on the inserted instrumentation. Our implementation also uses the CCured type inference engine to prove that some pointer dereferences are safe and hence do not require a check.

Latent check implementation. Memsafe assigns each instrumentation site a unique index within a global bit-vector. The instrumentation at that site only executes if its corresponding entry in the vector is set. The transformation generates inter-instrumentation dependencies and encodes them as a data-structure inside the resulting object file. The global bit-vector is loaded from a file at program startup. The run-time system uses the dependency information to enable all supporting checks after initializing the bit-vector from disk.

Our transformation generates fast-path/slow-path code, as described in Section 3. Each function decides whether to run its fast path or its slow path upon entering the function. The function can also switch between fast and slow paths at the beginning and end of every loop. Thus, for example, if the function is executing along its slow path and reaches a loop that contains no active instrumentation, then the function will temporarily switch to the fast path for the duration of the loop. The optimal placement of switching points depends on the program's run-time behavior, so we chose the above heuristic since it is likely to give a good pay-off for relatively few switching points. If a function contains no intrumentation, then we only generate a single path for it.

As with dependencies between instrumentations, there are also control-flow dependencies between switch points and instrumentations. Memsafe computes these dependencies at compile time and embeds them in the same dependency data structure as the inter-instrumentation dependencies. The run-time dependency resolution algorithm thus activates the correct set of switch points for any set of active instrumentations.

5 Evaluation

We implemented Memsafe as a source-to-source transformation using the CIL program analysis framework. Our evaluation focuses on two aspects of the transformation: correctness and performance.

Correctness. For backwards compatibility, a correct transformation should allow program executions that do not exhibit a memory error to execute normally. For security, a correct transformation should cause program executions with a memory error to halt as soon as the error occurs. Furthermore, we must verify that transformed code runs correctly with no checks enabled, all checks enabled, and arbitrary subsets of checks enabled.

Since memory errors taken from real-world programs can be brittle and highly-dependent on architecture and compiler details, we have chosen to evaluate the correctness and security of our transformation using a suite of simple test programs. Each test program accepts a command-line argument indicating whether it should execute code with a buffer overflow. The test programs are designed so that, when compiled with a normal compiler, the buffer overflows are all silent and harmless. We run each of these programs through the Memsafe transformation and verify that the resulting executables all satisfy the following requirements

- With all checks disabled and with no buffer overflow, the program executes normally.
- When all checks are enabled and no buffer overflow occurs, the program completes execution normally.
- When all checks are enabled and a buffer overflow occurs, the program aborts on a run-time check.
- We then turn off all checks except the failing check from the previous test (and any supporting checks), and re-run the program with a buffer overflow. It must abort as before.
- Finally, we run the program several times with random subsets of 10%, 20%, 30%, and 40% of the run-time checks enabled, but with no buffer overflow, and confirm that the program always executes normally.

Our test-suite currently contains 54 different tests, including several hand-written tests, tests derived from programs written by the authors for un-related projects, and gzip 1.2.4, which contains a known buffer overflow. All tests pass. The benchmarks used for the performance analysis described below serve as further evidence of the correctness of our transformation.

	GCC	CIL	Memsafe	
			All off	All on
	Time	Ratio	Ratio	Ratio
bh	1.22	1	1.16	4.05
bisort	0.88	1.01	1.02	1.32
em3d	1.26	1.03	1.04	55.9
health	1.62	1.01	1.03	48.48
perimeter	0.61	1.06	1.05	1.05
power	0.94	1	1.21	1.52
treeadd	0.12	1.42	1.42	6.25
tsp	1.41	1	1	1
gzip	3.37	0.97	0.98	3.49
gunzip	0.62	0.84	1.06	3.39
Average	**N/A**	**1.034**	**1.097**	**12.645**

Table 1: Overhead of our transformation when compiled with optimization, i.e. gcc's -O3 mode.

Performance. Usually there will be no known vulnerabilities in an application, so users will run the application with all checks disabled. Thus overhead in this configuration is the most important. Occasionally, the user will have a single check activated, along with its supporting checks, because of a known vulnerability in the application. This overhead should be as low as possible, but it is less important than overhead with all checks off. Finally, during testing, the vendor or user may wish to run the application with all checks enabled. This overhead is not too important unless it is so high that the application is unusable. Thus, we measure the overhead of our prepatching transformation in these three configurations: all checks off, one check enabled, and all checks enabled.

Table 1 shows the overhead of our transformation on the Olden benchmark when all checks are disabled or all checks are enabled. The mst benchmark from the Olden suite stores out-of-bounds pointers in non-solid variables, and so is not supported by our transformation. The transformation appears to incur modest overhead (about 10%) when all checks are disabled. It is interesting to note that simply running the code through CIL can produce significant changes to the running time of a program – from over 15% faster to over 40% slower. Relative to the base overhead of CIL, memsafe adds only about 6.1% additional overhead.

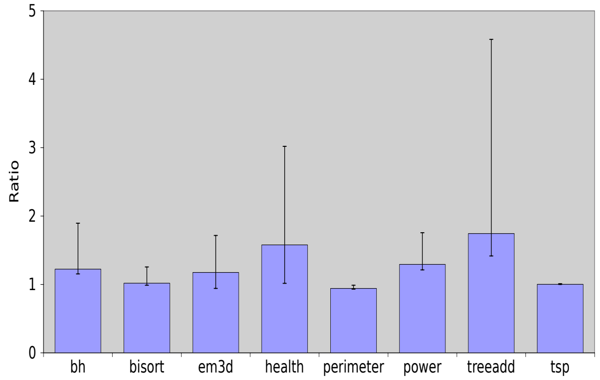

Figure 2: Memsafe overhead with one check enabled. The TSP error bars are present, but extremely small.

Table 1 shows that the overhead when all checks are enabled can be very high. Although lower overhead would be better, this is acceptable because the software is rarely run in this mode.

Figure 2 shows the overhead when a single check is activated in our benchmark programs. Since the overhead varies depending on which check is activated, we ran each benchmark 100 times, activating a randomly chosen check on each execution. We report the average execution time ratio, along with error bars indicating the 95% confidence interval. The benchmarks are compiled without GCC optimizations. Overheads can vary significantly depending on the activated check, but on average the overhead is about 25%. Although this is a moderately high overhead, the benefit is that, in the common case when all checks are off, the overhead is much lower, e.g. 6-10%, as shown in Table 1.

On average, benchmark executables created with Memsafe are 2.16 times larger than those created with GCC.

6 Conclusion

Pre-patched software turns the normal patching model on its head. By generating run-time checks in advance, but leaving them disabled until necessary, vendors can react quickly to newly-discovered bugs and worms without incurring a high run-time overhead.

Our prototype demonstrates that pre-patching is a feasible mechanism for dealing with low-level bugs, such as memory-safety errors in C. The techniques developed for our prototype can be used to create a pre-patching compiler that addresses other security bugs.

References

[1] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the 14th Annual Network & Distributed System Security Symposium*, 2007.

[2] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[3] J. Condit, M. Harren, S. McPeak, and etc. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming L anguage Design and Implementation*, June 2003.

[4] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In M.K. Kamkar and D. Byers, editors, *Proceedings of the Third International Workshop on Automated Debugging*, volume 2–9 of *Linköping Electronic Articles in Computer and Information Science*, pages 13–26, Linköping, Sweden, May 1997. Linköping University Press.

[5] Zhenkai Liang, R. Sekar, and D. C. DuVarney. Automatic Synthesis of Filters to Discard Buffer Overflow Attacks: A Step Towards Realizing Self-Healing Systems. In *Proceedings of the USENIX Annual Technical Conference (to appear)*, 2005.

[6] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.

[7] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, February 2004.

[8] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, August 2004.

[9] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[10] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. *SIGSOFT Software Engineering Notes*, 29(6):117–126, 2004.