An Architecture A Day Keeps The Hacker Away

An Architecture A Day Keeps The Hacker Away

David A. Holland, Ada T. Lim, and Margo I. Seltzer
Harvard University
{dholland,ada,margo}@eecs.harvard.edu

September 15, 2004

Abstract

System security as it is practiced today is a losing battle. In this paper, we outline a possible comprehensive solution for binary-based attacks, using virtual machines, machine descriptions, and randomization to achieve broad heterogeneity at the machine level. This heterogeneity increases the "cost" of broad-based binary attacks to a sufficiently high level that they cease to become feasible. The convergence of several recent technologies appears to make our approach achievable at a reasonable cost, with only moderate run-time overhead.

1 Insecurity

Today, security is a losing battle. There is always one more bug, and generally that means there is also always one more exploitable bug. To close these holes, administrators must keep up with a constant stream of patches and fixes; these are not always themselves entirely benign relative to the functioning of the systems they protect.

Worse, numbers are on the side of the hackers. Sooner or later after an exploitable bug is discovered, canned exploit scripts for the bug begin to circulate. Only one person need write such a script; once the script exists, any passing vandal (or automated worm) can use it to break into thousands of computers. This puts administrators, who have to patch machines individually or in small groups, at a huge disadvantage.

Some techniques have been developed to block whole classes of attacks: for example, StackGuard [5] effectively prevents a certain type of buffer-based attack; the more recent PointGuard [4] protects against a wider range of such attacks. Recent work in shepherded execution [10] has the potential to stop code-injection attacks and many flow-of-control attacks. However, most of these tools are quite specific in nature, and in many cases they amount to an arms race between exploit writers and security tool developers, with both sides becoming increasingly creative and clever. This means that while they help, they don't really offer a lasting solution to the underlying problem of security bugs.

2 Monoculture

Observe that most exploit scripts that circulate widely target only the most commonly deployed platforms. In the early 1990s this meant that most circulating exploits were for Sparcs running SunOS. Today it means that most exploits target Intel-architecture computers running either Windows or Linux. Exploits for the same bugs that target other platforms circulate much less readily.

Why is this? The short answer is that most exploits, particularly those for remotely-exploitable vulnerabilities, are not portable because they are *binary*.

They rely on specific machine-level characteristics of the target platform in order to work: word size, byte order, calling conventions, compiler peculiarities, program load addresses, even the instruction set itself. Different machine architectures, and, often, different operating systems on the same architecture, are slightly different in these respects, so that while all platforms may be vulnerable, any particular exploit must be customized for one specific platform. There are thus diminishing returns for targeting anything but the most common platforms. For example, the Morris worm attacked only two platforms, even in the relatively heterogeneous world of 1988 [12].

As the OS market continues to consolidate, the most common platforms become relatively more common still, and this problem worsens. Much has been written recently about the risks of *monoculture* [8]. As in biology, a certain degree of heterogeneity is healthy [6]: it provides protection for both the individual, because the individual may be resistant to any particular problem, and for the population as a whole, because any particular problem is unlikely to be able to affect everything at once.

Individual sites can leverage this principle to a certain degree by running exposed services on less common platforms, a practice the authors have personally found to be useful. However, doing so is an instance of security by obscurity: it does not scale, and successful exploit scripts might always appear. Furthermore, the supply of suitable platforms is limited.

3 My Own Private Architecture

Imagine instead that every site, or every host, were able to run on its own unique machine architecture.

Provided that each unique architecture is sufficiently different from all others, no canned exploit script would work on *any* target: every exploit would have to be written explicitly for the particular host it was attacking. This eliminates the economies of scale that hackers currently leverage: administrators still must patch machines individually, but hackers must target machines individually as well. Furthermore, it puts the so-called "script kiddies" out of business entirely: with no canned exploit scripts, breaking into systems by rote is no longer possible.

The possible advantages for the security and integrity of the Internet are enormous. So the question arises: how can this level of architectural diversity be deployed? It is obviously infeasible to invent millions of new machine architectures and fabricate them in hardware.

The answer, we argue, lies in combining two recently popular ideas: virtual machine monitors and machine descriptions. We envision a virtual machine monitor that runs standalone on commodity hardware and presents an appearance indistinguishable from a (perhaps slower) exotic machine implemented directly in hardware.

It seems to be generally accepted at present that a good virtual machine monitor is sufficient to contain attacks (and hackers) within the virtual machine and prevent direct attacks on the host. For example, Garfinkel et al. [7] recently proposed a system whose entire security basis rests on using a virtual machine monitor this way. Similarly, honeypots are routinely built using virtual machines [13].

Other recent work [2], as well as experience with JVMs, suggests that the overhead from the virtual machine can be made tolerable – although probably not negligible, particularly if the virtualized machine has properties that do not map well onto commodity hardware.

At the same time, recent work on machine descriptions has shown that it is possible to generate not just compiler backends, for which the technique is routine, but also virtual machine monitors, assemblers, linkers, debuggers, and other architecture-dependent tools, all from concise machine descriptions [11]. We believe that it is also practical to generate the architecture-dependent parts of a kernel and standard C library from machine descriptions. This claim is

discussed in more detail below.

Pulling all these pieces together, it becomes possible, in principle, to generate all the machine-dependent parts of a complete operating system, as well as the virtual machine monitor for running it, all from a concise machine description. Thus, installing a machine that uses its own unique machine architecture, given the description for that architecture, is as simple as recompiling the world.

In an open source world this is clearly a viable proposition: while installing by building the world takes longer than installing by copying binaries from CD, it is equally automatable and incurs only a one-time cost. Note, however, that even in a closed-source world, precisely the same principles can be made to apply: if software is shipped as Java byte code, Microsoft .NET byte code, or any other similar abstraction at a higher level than raw machine code, it can be compiled to run natively on the unique architecture.

4 Machine Description Space

A crucial part of this approach is that the space of possible machine descriptions be large enough to offer statistical protection against various kinds of attacks. This means that, among other things, it should not be feasible to mount an attack by exhaustively trying exploits for all possible machine types against a single target. Furthermore, the number of deployed systems sharing the same machine description should be small.

For the sake of argument, suppose 100 attacks can be made per second and we want, on average, to resist a week of continuous attack. This requires slightly under 2^{27} distinct machines. This is also probably enough to cover the deployed population. Note that this can be achieved with only 27 independent binary decisions about the architecture.

If possible, we would like this mechanism to protect against all reasonably foreseeable binary attacks. To this end, we will divide the set of binary attacks into two categories. The first of these is the *code injection attack*. This category includes any attack that inserts machine code into a target program, then persuades that program to execute that code. (Attacks that insert standalone executables into the system may fall into this category, if the executables are binaries; but since in general such executables may be scripts, such attacks are not strictly binary and we do not consider them.) Traditional buffer overflow attacks, `printf` format string attacks, and so forth are all code injection attacks.

An examination of the possibilities associated with tweaking the instruction set, and with varying the instruction coding, will show that there are thousands of bits of variation possible, far beyond the 27 we require. Thus we conclude that randomizing the architecture easily blocks code injection attacks. However, code injection can already be defeated by other means, such as applying an XOR pad to the instruction stream [9] or, in many cases, even the simple technique of disallowing execution from writeable memory.

We thus turn to the second category of binary attack, the *state corruption attack*. This category includes attacks that modify the state of a program in order to persuade it to perform actions it shouldn't, but that do not inject any code. This includes direct attacks on the flow of control, such as buffer overflow attacks that work by provoking a jump directly into the standard library, or more complex attacks on program data like many based on integer overflows and free-twice bugs. State corruption attacks are a much more challenging and interesting problem.

Getting enough bits of variation to block state corruption attacks appears feasible, but not entirely trivial. We suggest the following techniques as a beginning; more can probably be invented by creatively abusing the C standard.

1. The differing size of various operations with different instruction sets and instruction encodings generates variability in the layout of pro-

gram code. We expect this effect to be relatively small, because differences will tend to cancel out. Thus, somewhat arbitrarily, we assign it two bits of space.

2. The number of registers. This affects the layout of the stack and of code as different numbers of registers need to be saved and restored. There will probably be either 8, 16, 32, or 64 registers, as these are the powers of two in the useful range. This gives two bits of description space.

3. The machine byte and word sizes can be chosen from a list of possible combinations. It may prove necessary to use only 8-bit bytes and 32-bit or 64-bit words. However, in principle it is possible to use 9-bit or 10-bit bytes with 36-bit or 40-bit words, 40-bit words with 8-bit bytes, 16-bit bytes, 24-bit bytes that will hold Unicode natively, or other things. Doing so would be advantageous because such platforms would exhibit totally different behavior in the face of integer overflow bugs. Supposing eight viable possibilities, this gives another three bits of space.

4. There are normally only two endiannesses, but historically the VAX used a third. We can in fact use any possible byte ordering for words. The number of choices depends on the word size, but we shall suppose we get five bits of space on average.

5. The representation of signed integers gives us one bit of space, for the choice between two's complement and sign-magnitude. (The C99 standard apparently no longer allows one's complement.)

6. Stack direction (up or down) gives another bit.

7. Using one stack or two (splitting call and data stacks like a Forth machine) gives yet another bit.

8. Moving to less hardware-oriented properties, there is a wide variety of possible function-calling conventions, with different ordering, alignment, padding, registerization, stack adjustment, return value handling, and so forth. Calling conventions have been shown to be quite complex [1], and even with a fairly simple model there are probably at least 8 bits of description space to be used.

9. Likewise, various models can be used for alignment padding in stack frames and data structures, and possibly inserting small numbers of NOPs into code. This gives potentially another 8 bits of description space.

This adds up to 31 bits to defend against state corruption attacks. This is more than the 27 bits we need. However, it is not *much* more; some of those bits may not apply to some attacks. However, because both the figure of 27 bits and our analysis of the bits available are fairly conservative, we feel the description space is probably large enough to be useful, even against state corruption attacks.

Note that in addition to things that are, strictly speaking, part of the machine architecture, anything else that can be parameterized can be tweaked using similar techniques. This is a subject for future work.

One particularly radical possibility is to use a random character set in place of ASCII or Unicode. This would protect against exploits that carry well-known filenames (e.g., `/bin/sh`) and possibly against exploits that carry source code or scripts. However, this is likely to be excessively difficult to deploy.

5 Machine Descriptions for Kernels

We claim that it should be possible to generate the machine-dependent parts of a kernel, and C standard library, from machine description files. (Recall that the remaining tools we will need have already been addressed by other work [11].)

An examination of the machine-dependent directories of the NetBSD kernel reveals three categories

of machine dependencies. First, there are concepts that are equivalent to concepts that the compiler and toolchain have to know. This category includes things such as the sizes of standard types and the linker relocation codes; these are solved problems. Second, there are features where heterogeneity provides little benefit, like the way bus configuration is done or the basic way the processor handles exceptions. These would be research issues if generating kernels for *real* machines, but for current purposes they can be ignored. The third category consists of issues that we must address.

This third category in turn breaks down into three subcategories:

1. context switches, in their various forms (user-to-kernel, process-to-process, signal delivery, etc.) and their ramifications (trap frames, `struct siginfo`, etc.), because our various architectures will have different register sets;
2. small chunks of assembly code that will differ across architectures, such as spinlock handling;
3. and the virtual memory system, because our various architectures will have different word sizes and thus different address space sizes.

Context switches are all, we believe, readily generated given the list of registers in the architecture and various flags associated with them specifying their properties (callee-save, which is the stack pointer, etc.) – the code involves little more than reading and writing these registers to memory.

The small chunks of assembly code are probably either enumerable (there are only so many ways to implement spinlocks, for example), or are code generation problems more or less equivalent to compilation and can be handled along similar lines. (For example, the `_mcount` code used in profiling is an ordinary machine-independent function with special register handling requirements.)

The virtual memory system is a bigger issue; however, our research group has been working on this problem and we believe it is tractable even in the general case of wildly differing real MMUs. For the purposes of this problem, there is little value to having more than one basic MMU design that can vary slightly according to machine parameters. Parameterizing the machine-dependent VM machinery to support this should be quite straightforward.

(We are not aware of any existing work aiming to generate kernel components from machine descriptions apart from our own work in progress.)

6 Randomization

One final question remains: how do you generate the unique machine descriptions? While it may be sufficient to simply write them down, doing so is not necessarily a trivial undertaking.

Our idea is to generate them randomly. Given the size of the space of possible machines, as discussed above, random generation offers both unpredictability and a statistical approximation of global uniqueness; these are highly desirable properties. Using randomization to promote heterogeneity is, of course, not new; it was proposed by Forrest et al. in 1997 [6] and has been the basis of much work since. The significant point in this paper is not the use of randomization; it is the breadth and scope of it.

One obvious question is whether random generation of machine descriptions is even possible. Apart from the instruction set, the description space outlined above is framed in terms of either-or choices or choices among a small number of alternatives. These choices are, of course, easily randomized. Ramsey's machine descriptions [11] are descriptive, not algorithmic, and thus in principle randomizable as well.

There is also always a temptation to make machine descriptions Turing-complete. While we can avoid this for any new tools we develop, it may be a problem for existing tools. The machine descriptions used by `gcc` are essentially Lisp code, and in addi-

tion to the formal machine description a large quantity of architecture-specific C preprocessor macros are required. Directly generating all this randomly is a dubious proposition. Most likely, it will be necessary to determine how to generate a `gcc` machine description from a simpler, non-Turing-complete description of our own devising. This may be difficult.

Other randomization not tied to the machine description, such as the link-time or run-time address randomization proposed by Bhatkar et al. [3] and Xu et al. [14] can furthermore be used as a complementary approach to provide even more heterogeneity.

7 Major Caveat

This entire scheme depends on it *not* being possible to generate binary exploits from machine descriptions. If that turns out not to be true, this technique becomes merely another round in the arms race. Some benefits may still accrue if it is possible to keep your machine description secret.

It is not clear how practical it is or will be to generate exploits from machine descriptions.

8 Other Caveats

Despite the reassuring analysis above, it may turn out that in some cases it is possible to write an exploit that works for any target machine with a certain description property, or a certain set of them small enough to allow attacking a considerable number of machines at once. Note that even if this should be the case, it is still unlikely that *all* machines would be vulnerable: even if half the machines running random architectures are vulnerable, we still come out ahead.

Likewise, if the machine description space turns out not to be large enough, it may be possible to try all possible forms of an attack, or even write worms that do this in an automated fashion. Ten milliseconds per attempt may be unrealistically slow in this case.

It is not necessarily important to keep your machine description secret: even if you post it on your web page, you are still more or less immune to attack by worms and "script kiddies". To attack you, someone would have to target you explicitly: first download your machine description, then prepare a customized attack specifically against your machine. This assumes, of course, that the major caveat above does not become a problem. If it does, then not only do you need to keep your machine description secret, but there is another interesting catch: it may be possible to infer portions of your machine description, even remotely, by issuing partial attacks and observing the results. This could conceivably narrow the attack space, even against a completely unknown machine, enough to allow an attack to succeed in a short period of time.

It may turn out to be possible to attack the virtual machine monitor. Our technique will do nothing to prevent code injection or other attacks against the virtual machine monitor itself, if it should turn out to have suitable bugs. By making the virtual machine monitor relatively simple and small, it should be possible to keep the risk of such bugs low.

And, of course, this technique only protects against binary exploits. It does nothing to stop semantically-based attacks (such as `/tmp` symlink race conditions) or logic errors, and it will not prevent denial of service.

9 Other Comments

It is not necessarily required, or even desirable, for absolutely every deployed machine to be unique. A site with a large server farm could choose, at the risk of having a targeted attacker take over that entire server farm, to use only one architecture across the farm; this could help contain deployment costs.

On the other hand, it is also possible to aim for a certain degree of uniqueness in time: a truly paranoid site might rebuild with a fresh machine description every week, or, indeed, every day, just to rule out the possibility that someone might be preparing a targeted attack.

10 Challenges

There will be numerous challenges in attempting to build and deploy this system. Some have already been noted: working with `gcc`'s machine descriptions, for example. Others are not so obvious.

First, the toolchains and debuggers based on machine descriptions will need to move out of the lab and into production. This is potentially a large step.

Though we believe it perfectly feasible, generating kernel components from machine descriptions is likely to be a challenging research project.

Virtual machine technology as it stands achieves good performance for sane architectures. However, we are deliberately pursuing insane architectures. Efficiency will take work.

Vast amounts of both system and application code are bound to turn out not to be as portable as everyone thought. Even though mainstream 64-bit machines have been in the field for more than ten years, code still appears that assumes `sizeof(long)` $==$ `4`. Much more will break on the architectures proposed in this paper. Addressing this may turn out to be an extremely large project. (Arguably, however, it is worthwhile on its own merits.)

Relatedly, testing and debugging will become more interesting in this environment. One might argue that expecting code to work correctly on a brand new and entirely untested architecture is unreasonable. On the other hand, in practice, portable code is more robust, precisely because it has been tested under a range of varying circumstances. It is not obvious *a priori* which effect will dominate in the long run. Since each architecture is completely deterministic once generated, deterministic bugs will stay deterministic; this is to be distinguished from compile-time or run-time randomization, which makes debugging and testing a guessing game.

We will want our compilers to be more aggressive about identifying machine-dependent or undefined behavior; however, as optimizers grow smarter this is becoming increasingly important anyway.

11 Is It Worthwhile?

Given all the challenges, the question arises whether this idea is really worth pursuing, given that many existing techniques offer a large measure of protection without being anywhere near so intrusive.

The answer to this question comes in two parts. The first is technical: randomizing the entire architecture offers a markedly higher level of protection (over more limited techniques) against clever, unanticipated state corruption attacks. It is also, as a comprehensive approach, more robust from a systemic perspective than a patchwork of partial techniques.

The second, and perhaps more important, answer is social and environmental: in the long run, widespread adoption of architectual heterogeneity has the potential to change the security landscape. While one cannot realistically hope for all canned exploits, "script kiddies", and worms to go away, binary-based ones would. It is not clear that attacks based on logic errors, injecting portable script code, and so forth wouldn't take up the slack; however, it is not clear that such attacks can or would, either.

12 Conclusion

In this paper, we have proposed a technique that has the potential to radically alter the security landscape.

It has a number of possible drawbacks and limitations, but also has a considerable potential benefit. We believe it to be a viable idea worth pursuing, despite the amount of work involved.

References

[1] M. W. Bailey and J. W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of Principles of Programming Languages (POPL 95)*, pages 298–310, January 1995.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, October 2003.

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, DC, August 2003.

[4] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard™: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, DC, August 2003.

[5] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[6] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, October 2003.

[8] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Quarterman, and B. Schneier. Cyber insecurity: The cost of monopoly. Technical report, Computer & Communications Industry Association, 2003.

[9] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 272–280, Washington, DC, October 2003.

[10] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, San Francisco, Aug 2002.

[11] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 172–188, June 1998.

[12] D. Seeley. A tour of the worm. In *Proceedings of the 1989 Winter USENIX Conference*, January 1989.

[13] K. Seifried. Honeypotting with vmware - basics, 2002. Online. Internet. March 9, 2004. Available WWW: `http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.html`.

[14] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems*, Florence, Italy, October 2003.