(un) Smashing the Stack:

28 75 6e 29 53 6d 61 73 68 69 6e 67 20 54 68 65 20 53 74 61 63 6b 3a

Overflows, Countermeasures,

4f 76 65 72 66 6c 6f 77 73 2c 20 43 6f 75 6e 74 65 72 6d 65 61 73 75 72 65 73 2c

and the Real World

61 6e 64 20 74 68 65 20 52 65 61 6c 20 57 6f 72 6c 64

Shawn Moyer

Chief Researcher, SpearTip Technologies

http://www.speartip.net

{ b l a c k h a t } [at] { c i p h e r p u n x } [dot] { o r g }

0x00: Intro :: Taking the blue pill (at first).

My first exposure to buffer overflows, like much of my introduction to the security field, was while working for a small ISP and consulting shop in the 90's. Dave, who was building a security practice, took me under his wing. I was a budding Linux geek, and I confessed an affinity for Bash. After a brief lecture about the finer points of tcsh, Dave borrowed my laptop running Slackware, and showed me the Bash overflow in PS1, found by Razvan Dragomirescu.

This was a useful demonstration in that a simple environment variable would work to overwrite a pointer, though I immediately asked the importunate question of what good it did anyone to get a command shell to crash and then, well, run a command, in the context of the same user. I supposed if I ever encountered a restricted Bash shell somewhere, I was now armed to the teeth.

Just the same, I wanted to understand: how did those bits of shellcode get where they shouldn't be, and get that nasty "/bin/ls" payload to run?

Not too long after Dave's demonstration, I spent a lot of time puzzling over Aleph One, got my brain around things relatively well, and then rapidly went orthogonal on a career that rarely, if ever, touched on the internals of buffer overflows. I was far too busy over the next ten years or so (like most folks in InfoSec) building defenses and sandbagging against the deluge of remote exploits hitting my customers and employers. I spent my days and nights scouring BugTraq (later Vulnwatch and Full-Disclosure), writing internal advisories, firefighting compromises and outbreaks, and repeating the same mantra to anyone who would listen:

Patch early, and patch often.
Rinse, lather, repeat.

Service packs begat security rollups.
Security rollups begat Patch Tuesday.
Patch Tuesday begat off-cycle emergency updates.

Last week, the network admins rebooted my workstation three times. Seriously.

In the past few years, we seem to have found ourselves, as Schneier often points out, getting progressively worse and worse at our jobs. While aggressive code auditing of critical pieces of infastructure like Bind, Apache, Sendmail, and others may have reduced the *volume* of memory corruption vulnerabilities found in critical services in recent years, they haven't reduced the *severity* of the exposure when they are.

Of course, the client end is a minefield as well – email-based attacks, phishing, pharming, XSS, CSRF and the like have all shown that users are unfailingly a weak link, to say nothing of web application threats and the miles of messy client- and server-side issues with Web 2.0... Just the same, memory corruption vulnerabilities can lead to exploitation of even the best-educated, best-hardened, best-audited environments, and render all other protection mechanisms irrelevant.

The most recent proof that comes to mind, likely because I spent a very long week involved in cleanup for both private sector and some government sites, is Big Yellow. Say it with me: A

remote service. Listening on an unfiltered port on thousands of machines. Running with very high privileges. Vulnerable to a stack-based overflow.

Sound familiar?

In my caffeine-addled fog on an all-night incident response for this latest worm-of-the-moment, I asked myself: *Does this make sense?* Should we really be blaming vendors, or disclosure, or automation, or even the cardinal sin of failing to patch, for what ultimately comes down to a fundamental problem in error handling and memory allocation, described to me so succinctly by Dave all those years ago as “ten pounds of crap in a five pound bag”?

Recently, Jon Richard Moser of Ubuntu Hardened did an analysis of the first 60 Ubuntu Security Notices, and found that of these, around 81% were due to either buffer overflows, integer overflows, race conditions, malformed data handling, or a combination of all four. Moser believes that the aggregate of available proactive security measures in compilers, kernel patches, and address space protections available today could serve to obviate many, if not all of these vulnerabilities.

After a lot of digging, I think Moser may be right, though the devil, of course, is in the details.

0x01: When Dinosaurs roamed the Earth.

The first widely exploited buffer overflow was also what’s generally credited as the first self-replicating network worm, the response to which is covered in detail in RFC1135, circa 1988: “The Helminthiasis of the Internet”. A helminthiasis, for those without time or inclination to crack open a thesaurus, is a parasitic infestation of a host body, such as that of a tapeworm or pinworm. The analogy stuck, and all these years later it’s still part of the *lingua franca* of IT.

The Morris Worm was a 99-line piece of C code designed with the simple payload of replicating itself, that (intentionally or otherwise) brought large sections of the then-primarily research network offline for a number of days, by starving systems of resources and saturating network connections while it searched for other hosts to infect.

What’s relevant today about Morris is one of the vectors it used for replication: a stack-based overflow in the *gets()* call in SunOS’s fingerd. In his analysis in 1988, Gene Spafford describes the vulnerability, though he’s a bit closed-mouthed about the mechanics of how things actually worked:

> *The bug exploited to break fingerd involved overrunning the buffer the daemon used for input. The standard C library has a few routines that read input without checking for bounds on the buffer involved. In particular, the gets() call takes input to a buffer without doing any bounds checking; this was the call exploited by the Worm.*
>
> *The gets() routine is not the only routine with this flaw. The family of routines scanf/fscanf/sscanf may also overrun buffers when decoding input unless the user explicitly specifies limits on the number of characters to be converted. Incautious use of the sprintf routine can overrun buffers. Use of the strcat/strcpy calls instead of the strncat/strncpy routines may also overflow their buffers.*

What strikes me most about the above is that Spafford is still spot-on, nineteen years later. Unchecked input for misallocated strings or arrays, and the resulting ability to overwrite pointers and control execution flow, whether on the stack, the heap or elsewhere, remains a (mostly) solvable problem, and yet the exposure remains with us today.

After Morris, things were a bit quieter for awhile on the buffer overflow front. Rapid adoption of PC's, and the prevalence of nothing even resembling a security model for commodity operating systems, meant that the primary attack surfaces were boot sectors and executables, and for the rest of the 1980's virii were of substantially more scrutiny as an attack vector, for both defenders and attackers.

This isn't to say this class of vulnerabilities wasn't known or understood, or that Morris was the first to exploit them – in fact Nate Smith, in a paper in 1997, describes "Dangling Pointer Bugs", and the resulting "Fandango on Core" as being known of in the ALGOL and FORTRAN communities since the 1960's!

As has widely been stated, as soon as alternatives to writing code directly to hardware in assembler became readily available, the abstraction has created exposure. Of course, I'd be remiss if I didn't point out that for nearly as long, a move to type-safe or even interpereted languages has been suggested as the best solution.

Just the same, let's accept for now that the massive installed base of critical applications and operating systems in use today that are developed in C and C++ will make this infeasible for many, many years to come. Also, as Dominique Brezinski pointed out in a recent BlackHat talk, even an interpereted language, presumably, needs an interpereter, and overflows in the interpereter itself can still lead to exploitation of code, safe types, bounds-checking, and sandboxing notwithstanding.

0x02: Things get interesting.

In February of 1995, Thomas Lopatic posted a bug report and some POC code to the Bugtraq mailing list.

> *Hello there,*
>
> *We've installed the NCSA HTTPD 1.3 on our WWW server (HP9000/720, HP-UX 9.01) and I've found that it can be tricked into executing shell commands. Actually, this bug is similar to the bug in fingerd exploited by the internet worm. The HTTPD reads a maximum of 8192 characters when accepting a request from port 80. When parsing the URL part of the request a buffer with a size of 256 characters is used to prepend the document root (function strsubfirst(), called from translate_name()). Thus we are able to overwrite the data after the buffer.*

The unchecked buffer in NCSA's code to parse GET requests could be abused due to the use of strcpy() rather than strncpy(), just as described by Spafford in his analysis of the Morris worm seven years earlier. He included some example code that wrote a file named "GOTCHA" in the server's /tmp directory, after inserting some assembler into the stack.

US-CERT recorded a handful of buffer-overflow-based vulnerabilities in the years since Morris, but what made a finding like Lopatic's so relevant was the rapid adoption of NCSA's httpd, and the growth of the Internet and its commercialization. This really was a whole new (old) ballgame. The ability to arbitrarily execute code, on any host running a web server, from anywhere on the Internet, created a new interest in what Morris' stack scribbling attack a number of years ago had already proven: memory corruption vulnerabilities were a simple and effective way to execute arbitrary code remotely, at will, on a vulnerable host.

In the next two years, Mudge released a short paper (which he described as really a note to himself) on using GCC to build shellcode without knowing assembly, and how to use gdb to step through the process of inserting code onto the stack.

Shortly after, Aleph One's seminal work on stack-based overflows expanded on Mudge, and provided the basis for the body of knowledge still relevant today in exploiting buffer overflows. It's hard (if not impossible) to find a book or research paper on overflows that doesn't reference "Smashing the Stack for Fun and Profit", and with good reason.

Aleph One's paper raised the bar, synthesizing all the information available at the time, and made stack-based overflow exploit development a refinable and repeatable process. This is not to say that the paper created the overflow problem, and almost certainly the underground had information at the time to rival that available to the legitimate security community. While in some ways kicking off the disclosure debate, what "Smashing the Stack" ultimately provided was a starting point for clearly understanding the problem.

Overflows began to rule the day, and in the late 90's a number of vulnerabilities were unearthed in network services, including Sendmail, mountd, portmap and Bind, and repositories of reusable exploit code like Rootshell.com and others became a source of working exploits for unpatched services for any administrator, pen-tester (and yes, attacker) with access to a Linux box and a compiler.

While other classes of remotely exploitable bugs were of course found during this time and after, it's fair to say that Crispin Cowan was accurate in 1998 when he referred to overflows as "the vulnerability of the decade". In 2002, Gerhard Eschelbeck of Qualys predicted another ten years of overflows as the most common attack vector. Can we expect the same forecast in 2012?

0x03: Fear sells.

For the most part, the "decade of buffer overflows" did little to change the reactive approach to vulnerabilities systemic to our field. With some notable of exceptions, while exploitation of memory corruption vulnerabilities became incredibly refined ("Point. Click. Own."), the burgeoning (now, leviathan) security industry as a whole either missed the point or, if you're of a conspiratorial bent, chose to ignore it.

Compromises became selling tools for firewall and IDS vendors, with mountains of security gear stacked like cordwood in front of organizations' ballooning server farms, and these, along with the DMZ and screened subnet approach, allowed the damage from exploitation to be contained, if not prevented.

Fortunes were made scanning for patchlevels, and alerting on *ex post facto* exploitation. Consultants built careers running vulnerability scanners, reformatting the results with their letterhead, and delivering the list of exploitable hosts (again, often due to memory corruption vulnerabilities in network services), along with a hefty invoice, to the CIO or CSO.

The mass of the security industry simply adopted the same model it had already refined with antivirus – signatures for specific attacks, and databases of vulnerable version numbers, for sale on a subscription basis. None of this addressed the fundamental problem, but it was good business, and like antivirus, if an organization kept their signatures updated and dedicated an army of personnel to scan and patch, they could at least promise some semblance of safety.

0x04: Yelling "theater" in a crowded fire.

While the march of accelerated patch cycles and antvirus and IDS signature downloads prevailed, a small but vocal minority in the security community continued to search for other solutions to the memory corruption problem.

Ultimately many of these approaches either failed or were proven incomplete, but over time, the push and pull of new countermeasures and novel ways to defeat them has refined these defenses enough that they can be considered sound as a stopgap that makes exploitation of vulnerable code more difficult, though of course not impossible.

The refinement of memory corruption attacks and countermeasures shares a lot with the development of cryptosystems: an approach is proposed, and proven breakable, or trustworthy, over time. As we'll see later, like cryptography, the weaknesses today seem to lie not in the defenses themselves, but in their implementation. Because so many different approaches have been tried, we'll focus on those that are most mature and that ultimately gained some level of acceptance.

0x05: Data is data, code is code, right?

The concept is beguiling: in order for a stack-based overflow to overwrite a return pointer, a vulnerable buffer, normally reserved for data, must be stuffed with shellcode, and a pointer moved to return to the shellcode, which resides in a data segment. Since the code (sometimes called "text") segment is where the actual instructions should reside on the stack, a stack-based overflow is by definition an unexpected behavior.

So, why not just create a mechanism to flag stack memory as nonexecutable (data) or executable (code), and simply stop classic stack-based overflows entirely? In the POSIX specification, this means that a given memory page can be flagged as PROT_READ and PROT_EXEC, but not PROT_WRITE and PROT_EXEC, effectively segmenting data and code.

SPARC and Alpha architectures have had this capability for some time, and Solaris from 2.6 on has supported globally disabling stack execution in hardware. 64-bit architectures have a substantially more granular paging implementation, which makes this possible much more trivially – this is what prompted AMD to resurrect an implementation of this in 2001 with their "NX" bit, referred to as "XD" (eXecute Disable) by Intel on EM64T.

Software-based emulation on 32-bit architectures typically requires a “line in the sand” approach, where some memory range is used for data, and another for code. This is far less optimal, and may be possible to circumvent under specific conditions. With hardware-based nonexecutable stack features now widely available, this will become less of an issue over time, but for now, software emulation is better than no protection at all.

Historically, execution on the stack had been expected in some applications – called trampolining, the somewhat cringeworthy process of constructing code on the fly on the stack can yield some performance and memory access benefits for nested functions. In the past, a nonexecutable stack has broken X11, Lisp, Emacs, and a handful of other applications. With the advent of wider adoption of NX, and “trampoline emulation” in software, this is no longer as much of an issue, though it delayed adoption for some time.

Solar Designer built the first software noexec implementation for the Linux kernel, in 1997. When it was proposed for integration into the kernel mainline, it was refused for a number of reasons. Trampolines, and the work required to make them possible, was a large factor. In a related thread on disabling stack execution, Linus Torvalds also gave an example of a return-to-libc attack, and stated that a nonexecutable stack alone would not ultimately solve the problem.

> *In short, anybody who thinks that the non-executable stack gives them any real security is very very much living in a dream world. It may catch a few attacks for old binaries that have security problems, but the basic problem is that the binaries allow you to overwrite their stacks. And if they allow that, then they allow the above exploit.*
>
> *It probably takes all of five lines of changes to some existing exploit, and some random program to find out where in the address space the shared libraries tend to be loaded.*

Torvald’s answer was prescient, and in recent years the most common approach to defeating hardware and software non-executable stack has been return-to-libc. On Windows, Dave Maynor also found that overwriting an exception handler or targeting the heap was effective, and Krerk Piromposa and Richard Embody noted that a “Hannibal” attack, or multistage overflow, in which a pointer is overwritten to point to an arbitrary address, and then shellcode is written to the arbitrary address in the second stage, could succeed. In all of these cases, data segments on the stack were not replaced with code, and so the read-exec or read-write integrity remained intact.

Still, Solar’s patch gained adoption among security-centric Linux distributions, and it offered some level of protection, if only by obscurity – most distributions of Linux had fully executable stacks, so typical exploits in wider use would fail on system using the patchset.

Over time, the inarguability of a simple protection against an entire class of overflows led to the nonexecutable stack being ubiquitous. Today, WinXP SP2, 2003, and Vista have software-based nonexecutable stacks and integrate with hardware protection on 64-bit platforms, as does Linux (via PaX or RedHat’s ExecShield), OpenBSD with W^X, and even (on Intel) MacOS X.

Outside of the use of other classes of overflows, such as writing to the heap, or ret-to-libc, likely the key issue with stack protection on any platform is the ability to disable it at will. The mprotect() function on Linux / Unix and VirtualProtect() in Windows allow applications to ask for stack execution at runtime, and opt out of the security model. Microsoft’s .NET JIT compiler, Sun’s JRE, and other applications that compile code at run-time expect to create code on the stack, so these may become an area of greater scrutiny in the future.

Certainly nonexecutable stacks are only a small part of the solution, and opt-out with mprotect() and VirtualProtect() give developers the ability to override them, but they are computationally inexpensive, and a worthy part of a larger approach.

0x06: The canary in the coalmine.

Crispin Cowan's StackGuard, released in 1997, was the first foray into canary-based stack protection as a mechanism to prevent buffer overflows. The approach was simple: place a "canary" value into the stack for a given return address, via patches to GCC, in *function_prologue*. On *function_epilogue*, if a change to the canary value was detected, the canary checks called exit() and terminated the process.

Cowan found that StackGuard was effective at defending against typical stack-based overflows in wide use at the time, either stopping them entirely, or creating a Denial of Service condition by causing the service to exit.

After StackGuard's initial release, Tim Newsham and Thomas Ptacek pointed out two issues in the implementation, less than 24 hours later. The problem was in the canary value's lack of randomization. If a guessable or brute-forceable canary was the only protection in place, the defense was only as good as the canary. So, either guessing the canary, or finding a way to read the canary value from memory, would render the defense void.

But even with a stronger canary value, the larger weakness of protecting only the return address remained. While the return address is one of the most effective and common targets in exploiting an overflow, it's by no means the only one. Essentially, any other area in memory was unprotected, so as long as the canary was intact, the injected shellcode still ran.

Originally introduced in Phrack 56 by HERT, an effective approach was demonstrated – writing "backward" in specific cases via an unbounded strcpy() could bypass the protection. The Phrack 56 article also proved exploitability of the same weaknesses in the canary value Newsham and Ptacek had already pointed out. This led to the adoption of a more robust approach to the canary value, and an XOR'd canary of a random value and the return address was eventually adopted in future versions. Gerardo Richarte of Core Security also demonstrated that writes to the Global Offset Table, "after" the return address, as well as overwrites of frame pointers and local variables, would still lead to code execution.

Hiroaki Etoh's ProPolice built on StackGuard's canary concept, but matured the approach much further, and created a full implementation that added canaries (Etoh prefers the term "guard instruments") for all registers, including frame pointers and local variables, and also reordered data, arrays, and pointers on the stack to make overwriting them more difficult: if pointers and other likely targets are not near data in memory, it becomes much more difficult to overwrite a given buffer and move the pointer to the supplied shellcode.

In 2004, Pete Silberman and Richard Johnson used John Wilander's Attack Vector Test Platform to evaluate ProPolice and a number of other overflow protection methods, and found ProPolice effective at stopping 14 of the 20 attack vectors tested by AVTP. ProPolice's primary weaknesses were in not protecting the heap and bss, and in not protecting smaller arrays or buffers.

ProPolice was accepted for inclusion with GCC 4.1, and was included in OpenBSD and Ubuntu as a backport to GCC 3.x. With 4.1 integration, it's now available in every major Linux and most

Unix distributions, and each of the BSD's. Microsoft also integrated a variant of XOR canaries and a limited level of stack reordering into WinXP SP2 and Windows 2003 and Vista. It's extremely important to note that compiler flags for what protections are enabled need to be set to take advantage of ProPolice on any platform, and are generally not enabled by default. On Linux / Unix, the "—fstack-protector and –fstack-protector-all" flags must be set, and on Windows, applications need to be compiled with /GS to gain ProPolice-like functionality.

0x07: /dev/random pitches in to help

In 2001, the PaX team introduced ASLR, or address space layout randomization, as part of the PaX suite of security patches to the Linux kernel. ASLR in a number of forms was also introduced into OpenBSD around roughly the same time, and due to some contention in these two camps over a number of topics, it's best to say that a bit of credit belongs to both, though I'm sure they shared a collective sigh when Microsoft introduced it five years later in Vista, to much fanfare and fawning in the IT press.

In general, ASLR randomizes memory allocation so that a given application, kernel task or library will not be in the same address with (hopefully) any level of predictability. This aims to make reusable exploits tougher to develop, as addresses to be targeted for example in a return-to-libc attack, like the address of the system() call, will not be in the same location on multiple machines.

Like attacks on TCP sessions with sequential ISN's, which made IP spoofing relatively trivial, an exploit usually needs a known, predicatable address to target. By randomizing memory in the kernel, stack, userspace, or heap, ASLR aims to make exploits less successful.

In practice, like StackGuard's canary value, if ASLR's randomization is weak, it becomes trivial to break. This is especially true for services that fork and respawn, such as Apache, or any service running inside a wrapper application that restarts it upon failure.

Hovav Schacham of Stanford (now UCSD) and a group of other researchers found that by enumerating offsets of known libc functions by a series of unsuccessful attempts – their example used usleep(), but any widely-available function with a known offset would work – they could effectively brute-force ASLR randomization through a series of failed overflow attempts on a respawning service, crashing it numerous times but eventually successfully exploiting the service and returning to libc.

With this method, Shacham was able to compromise a vulnerable ASLR-protected Apache via brute force in around 200 seconds. Since 32-bit ASLR implementations use far less entropy than 64-bit, it was noted that 64-bit ASLR would be substantially more time-consuming to defeat. Also, Shacham's scenario presumes a service that restarts rather than simply crashes, so detecting a repeated number of failures (which the PaX team also recommends) would render the attack a denial of service rather than code execution.

Most other methods of defeating ASLR work in a similar way: if an offset of a known-sized function can be obtained, return-to-libc is possible. In Phrack 59, "Tyler Durden" (a pseudonym and tip-of-the-hat to the film Fight Club) used format string vulnerabilities as an example to disclose addresses on a PaX-enabled system. Ben Hawkes of SureSec presented a method he called "Code Access Brute Forcing" at RuxCon in the past year that used, like Shacham, a series of unsuccesful reads to map out memory in OpenBSD.

On the Microsoft front, Ollie Whitehouse of Symantec performed a series of regression tests of Vista's ASLR, and found it to be substantially weaker on the heap and process environment blocks (PEBs) than in other areas, and that heap randomization was actually better using ANSI C's *malloc()* than MS's recommended *HeapAlloc()*.

Since even in the best cases Vista's ASLR is weaker than that of most other implementations in terms of bits of entropy, it seems likely that derandomization attacks like Shacham's will be effective to some extent. Additionally, like stack protection, Microsoft allows applications to opt-in or opt-out, which means for some apps protections may not be consistent.

If sufficiently randomized, and if not readable in some other way such as via format string bugs or other information leakage, ASLR does still present a substantial barrier to heap overflows and return-to-libc attacks. This is especially true if all applications are built as PIC or PIE (Position-Independent Executables|Code), which make it possible for running applications, in addition to libraries and stack or heap memory, to load at less predictable locations.

0x08: How about just fixing the code?

For some time, extensive code review has been posited as the best route to securing vulnerable applications. The OpenBSD project in particular has spent a number of years aggressively working to identify security bugs as part of the development process. After a number of remote vulnerabilities were still found in released code, Theo DeRaadt, long a proponent of pure code review and secure-by-default installations as the best approach to security, famously altered his position and began implementing a number of stack and heap protection measures as well as ASLR and other mechanisms to make overflow exploitation more difficult.

Still, fixing vulnerabilities in code before they become exposures is without question the most effective route, and software developers are far more aware today than in previous years of the importance of integrating security review into the development lifecycle, using both manual review and static code analysis.

In GCC, the RedHat-developed FORTIFY_SOURCE extension looks for a number of exploitable conditions at compile time, and when paired with its glibc counterpart, can identify if a buffer with a defined length is mishandled and stop execution, by replacing oft-abused functions with their checking counterparts. FORTIFY_SOURCE will also warn for conditions it deems suspect but cannot protect. OpenBSD has taken similar steps by replacing commonly exploited functions like strcat() / strcpy() with fixed-size alternatives.

A number of vendor products also use automated code analysis to identify security holes. Recently the US Department of Homeland Security invested $1.25 million in a joint project between Stanford, Coverity, and Symantec to search Open Source projects for security bugs. In 1999, as part of its security push, Microsoft purchased code analysis toolmaker Intrinsa outright, and made static analysis an integrated part of their QA process.

0x09: The sum of the whole is partially great.

If security in depth is the sum of a number of imperfect controls, then the controls described here should all certainly qualify. Each has been proven insufficient in a vaccuum, but the aggregate of a nonexecutable stack, ASLR, canary-based protection of pointer values, and static code analysis should still serve to create an environment that is more hostile to overflows than what has previously been available... The key weaknesses now seem to be in a lack of consistency of adoption and implementation.

A remotely-exploitable stack-based overflow in ANI cursor handling in Vista, found by Alexander Sotirov, was due in part to Visual Studio's /GS checks not protecting buffers that write to structures rather than arrays. Also, as NX and DEP have become more ubiquitous, heap exploitation and other alternatives have gained renewed interest as well.

OpenBSD elected to omit kernel stack randomization, though it was adopted by PaX, due to questions about whether it broke POSIX compliance. In recent months OpenBSD was found vulnerable to a number of vulnerabilities in the kernel – one such example is being presented this year at BlackHat. While I'm sure the OpenBSD camp will have an alternative answer, it seems to me that a randomized kstack might have at least raised the bar a bit.

OS X has benefited from relative obscurity for some time, but with increasing marketshare and minimal overflow protection beyond an optional integration with NX, it's likely to become an attractive target – attacks refined a number of years ago are relatively trivial to implement, when compared to exploiting the same bugs on other platforms.

In time, as always, new attacks will create new countermeasures, and the security ecosystem will continue to evolve, in fits and starts, as it always has, from RFC1135, to Aleph One, and on.

// Notes and references

Thanks for reading. I hope this paper was helpful to you. It's the result of my attempt to better understand what protections were out there for my own systems and those under my care, and to get all of this information in one spot, which is something I couldn't find a year ago when I got interested in this topic.

Here's a starting point for some of the information referenced in this paper. The BH CD also contains copies of these and a lot of other supporting material. In addition to these, I'd highly recommend reading Jon Erickson's excellent *Hacking: The Art of Exploitation*, and *The Shellcoder's Handbook*, edited by Jack Koziol.

– shawn

NX Bit, PaX, and SSP on Wikipedia
http://en.wikipedia.org/wiki/NX_bit
http://en.wikipedia.org/wiki/PaX
http://en.wikipedia.org/wiki/Stack-smashing_protection

PaX: The Guaranteed End of Arbitrary Code Execution
Brad Spengler
http://grsecurity.net/PaX-presentation.ppt

PaX documentation repository:
http://pax.grsecurity.net/docs/

Edgy and Proactive Security
John Richard Moser
http://www.nabble.com/Edgy-and-Proactive-Security-t1728145.html

What's Exploitable?
Dave LeBlanc
http://blogs.msdn.com/david_leblanc/archive/2007/04/04/what-s-exploitable.aspx

On the Effectiveness of Address-Space Layout Randomization
Shacham *et al.*
http://crypto.stanford.edu/~dabo/abstracts/paxaslr.html

Defeating Buffer-Overflow Protection Prevention Hardware
Piromposa / Enbody
http://www.ece.wisc.edu/~wddd/2006/papers/wddd_07.pdf

ByPassing PaX ASLR
"Tyler Durden"
http://www.phrack.org/archives/59/p59-0x09.txt

Johnson and Silberman BH talk on Overflow Protection Implementations
http://rjohnson.uninformed.org/blackhat/

Exploit mitigation techniques in OBSD
Theo DeRaadt
http://www.openbsd.org/papers/ven05-deraadt/index.html

Ubuntu USN analysis listing type of exploit (45% buffer overflows)
John Richard Moser
https://wiki.ubuntu.com/USNAnalysis

Crispin Cowan's StackGuard paper,
USENIX Security 1998
http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan_html/cowan.html

Detecting Heap Smashing Attacks through Fault Containment Wrappers:
http://ieeexplore.ieee.org/iel5/7654/20915/00969756.pdf

ContraPolice: a libc Extension for Protecting Apps from Heap-Smashing attacks
http://synflood.at/papers/cp.pdf

Effective protection against heap-based buffer overflows without resorting to magic
Younan, Wouter, Piessens
http://www.fort-knox.be/files/younan_malloc.pdf

l0t3k site with lots of linkage on BoF's
http://www.l0t3k.org/programming/docs/b0f/

How to write Buffer Overflows
Peter Zaitko / Mudge
http://insecure.org/stf/mudge_buffer_overflow_tutorial.html

Defeating Solar Designer's NoExec stack patch
http://seclists.org/bugtraq/1998/Feb/0006.html

Solar Designer / Owl Linux kernel patchset
http://openwall.com/linux/

Theo's hissy fit justifying ProPolice in OBSD to Peter Varga
http://kerneltrap.org/node/516

Stack-Smashing Protection for Debian
http://www.debian-administration.org/articles/408

IBM ProPolice site:
http://www.trl.ibm.com/projects/security/ssp/

Four different tricks to bypass StackGuard and StackShield
http://www.coresecurity.com/index.php5?module=ContentMod&action=item&id=1146

Smashing the Stack for Fun and Profit
Elias Levy / Aleph One
http://www.phrack.org/archives/49/P49-14

Stack Smashing Vulnerabilities in the Unix Operating System
Nathan P. Smith
http://community.corest.com/~juliano/nate-buffer.txt

RFC 1135
http://www.faqs.org/rfcs/rfc1135.html

Gene Spafford's analysis of the Morris Worm
http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf