Technical Report IR-CS-73

Memory Errors: The Past, the Present, and the Future

Victor van der Veen[†], Nitish dutt-Sharma[†], Lorenzo Cavallaro[†,*] Herbert Bos[†]
[†]The Network Institute, VU University Amsterdam
[*]Royal Holloway, University of London

***Abstract*—Memory error exploitations have been around for over 25 years and still rank among the top 3 most dangerous software errors. Why haven't we been able to stop them? Given the host of security measures on modern machines, are we less vulnerable than before, and can we expect to eradicate memory error problems in the near future? In this paper, we present a quarter century worth of memory errors: attacks, defenses, and statistics. A historical overview provides insight in past trends and developments, while an investigation of real-world vulnerabilities and exploits allows us to speculate on the significance of memory errors in the foreseeable future.**

I. Introduction

Memory errors in C and C++ programs are among the oldest classes of software vulnerabilities. To date, the research community has proposed and developed a number of different approaches to eradicate or mitigate memory errors and their exploitation. They range from safe languages that remove the vulnerabilities entirely [1], [2], to bounds checkers that perform runtime checks for out-of-bounds accesses [3], [4], [5], [6]. They also include lightweight countermeasures that prevent certain memory locations to be overwritten [7], [8], detect code injections at early stages [9] or prevent attackers from finding [10], [11], using [12], [13], or executing [14], [15] injected code.

Despite more than two decades of independent, academic, and industry-related research, such flaws still undermine the security of our systems. Even if we consider only classic buffer overflows, this class of memory errors has been lodged in the top-3 of the CWE SANS top 25 most dangerous software errors for years [16]. Experience shows that attackers, motivated by profit rather than fun [17] have been effective at finding ways to circumvent protective measures [18], [19]. Many attacks today start with a memory corruption that provides an initial foothold for further infection.

Even so, it is unclear how much of a threat these attacks remain if all our defenses are up. In two separate discussions among PC members in two of 2011's top-tier venues in security, one expert suggested that the problem is mostly solved as "dozens of commercial solutions exist" and research should focus on other problems, while another questioned the significance of our research efforts, as they clearly "did not solve the problem". So which is it? The question of whether or not memory errors remain a significant threat in need of renewed research efforts is important and the main motivation behind our work.

To answer it, we study the memory error arms-race and its evolution in detail. Our study strives to be both comprehensive and succinct to allow for a quick but precise look-up of specific vulnerabilities, exploitation techniques or countermeasures. It consolidates our knowledge about memory corruption to help the community focus on the most important problems. To understand whether memory errors remain a threat in the foreseeable future, we back up our investigation with an analysis of statistics and real-life evidence. While some papers already provide descriptions of memory error vulnerabilities and countermeasures [20], we provide the reader with a *comprehensive bird-eye view* and *analysis* on the matter. This paper aims to be *the* reference on memory errors.

To this end, we first present (Section II) an overview of the most important studies on and organizational responses to memory errors: the first public discussion of buffer overflows in the 70s, the establishment of CERTs, Bugtraq, and the main techniques and countermeasures. Our discussion blends academic, industry and underground-driven research for completeness, importance, and impact of the information. Like Miller et al. [21], we use a compact timeline to drive our discussion, but categorize events in a more structured way and based on a branched timeline. For instance, we have not followed the classic division between OS- and compiler-enforced protections [22]. Conversely, we strive to focus on memory error-related history and facts. Doing so helps the reader navigate through the dense and prolific maze of memory error-related topics (with the ability to zoom in and out) and contributes to a timeline-driven discussion of the key events. Branches of the timeline are the topic of detailed discussion in Sections III–IX.

Second, we present a study of memory errors statistics, analyzing vulnerabilities and exploit occurrences over the past 15 years (Section X). Interestingly, the data show important fluctuations in the number of *reported* memory error vulnerabilities. Specifically, vulnerability reports have been dropping since 2007, even though the number of exploits shows no such drop. A tentative conclusion, drawn in Section XI, is that memory errors are unlikely to lose much significance in the near future and that perhaps it is time adopt a different mindset—one where malicious computations, often as a result

of successful memory error exploitations, should be expected to take place eventually—necessitating further work on containment techniques.

II. A HELICOPTER VIEW OF MEMORY ERROR HISTORY

A memory error occurs when an object accessed using a pointer expression is different from the one intended. A spatial memory error occurs when a pointer pointing outside the bound of its referent is dereferenced. Spatial memory errors include dereferences of uninitialized pointers and non-pointer data, and valid pointers used with invalid pointer arithmetic where buffer overflows represent the classic example. Conversely, a temporal memory error occurs when a the program dereferences a pointer to an object that no longer exists. Representatives examples are dangling pointers and double frees, as discussed in Section V. The core history of memory errors, their exploitations, and main defenses techniques can be summarized by the branched timeline of Figure 1.

Memory errors were first publicly discussed in 1972 by the Computer Security Technology Planning Study Panel [23]. However, it was only after more than a decade that this concept was further developed. On November 2, 1988, the Internet (or Morris) Worm developed by Robert T. Morris abruptly brought down the Internet [24]. The Internet Worm exploited a number of vulnerabilities, including memory error-related ones.

In reaction to this catastrophic breach, the first Computer Emergency Response Team Coordination Center (CERT/CC) was then formed [25]. CERT/CC's main goal was to collect user reports about vulnerabilities and forward them to vendors, which would have taken the appropriate decision. In addition, the Morris Worm helped to bring memory errors to attention of the research community. Miller et al. published an empirical study of the reliability of UNIX utilities in which they provide evidence of how insecure systems were at that time [26].

In response to the lack of useful information about security vulnerabilities, Scott Chasin started the Bugtraq mailing list in November 1993. At that time, many considered the CERT/CC useless, vendors did little to help and administrators had to wait years before patches for security vulnerabilities were provided. In contrast, Bugtraq offered practitioners an effective tool to *publicly* discuss vulnerabilities and possible fixes, without relying on vendors' responsiveness. Such information could then be used to patch vulnerable systems quickly [27].

In 1995, Thomas Lopatic boosted interest in memory errors even more, describing a step-by-step exploitation of an error in the NCSA HTTP daemon [28]. Shortly after, Peiter Zatko (Mudge) released a private note on how to exploit the now classic memory errors: stack-based buffer overflows [29]. So far, nobody really discussed memory error countermeasures, but after Mudge's notes and the better-known document by Elias Levy (Aleph One) on stack smashing [30], discussions on memory error and protection mechanisms proliferated.

The introduction of the non-executable (NX) stack opened a new direction in the attack-defense arms-race as the first countermeasure to address specifically code injection attacks in stack-based buffer overflows. Alexander Peslyak (Solar Designer) released a first implementation of an NX-like system, StackPatch [31], in April 1997. We discuss NX in Section III.

A few months later, in January 1998, Cowan et al. proposed placing specific patterns (canaries) between stack variables and a function's return address to detect corruptions of the latter [7]. We discuss canary-based defenses in Section IV.

After the first stack-based countermeasures, researchers started exploring other areas of the process address space–specifically the heap. In early 1999, Matt Conover and the w00w00 security team were the first to describe heap overflow exploitations [32]. We discuss heap attacks in Section V.

On September 20, 1999, Tymm Twillman introduced format string attacks. In his Bugtraq post, he describes an exploit against ProFTPD [33]. Format string exploits became popular in the next few years and we discuss them in Section VI.

The idea of adding randomness to prevent exploits from working (e.g., in StackGuard) was brought to a new level with the introduction of Address Space Layout Randomization (ASLR) by the PaX Team in July 2001. The first release randomized only stack locations to hamper exploits from finding a suitable location in memory to jump to (i.e., to execute code). Randomization became a hot topic in the following years and we discuss the various types of ASLR and its related attacks in Section VII.

Around the same time as the introduction of ASLR, another type of vulnerability, the `NULL` pointer dereference, was disclosed in May 2001 [34]. Many assumed that such dereferences were unlikely to cause more harm than a simple denial of service attacks. In 2008, however, Mark Dowd showed that `NULL` pointer dereferences could be used for arbitrary code injection as well [35]. We discuss `NULL` pointer dereferences in more detail in Section VIII.

III. NON-EXECUTABLE STACK

Stack-based buffer overflows [30] are probably the most common and well-understood memory error vulnerabilities. They occur when a stack buffer overflows and overwrites adjacent memory regions. The most common way to exploit them is to write past the end of the buffer until the function's (saved) return address is reached. The corruption of this code pointer permits to execute arbitrary code when the function returns. A non-executable stack prevents such attacks by marking bytes of the stack as non-executable. Any attempt to execute the injected code triggers a program crash. The first non-executable stack countermeasure was proposed by Alexander Peslyak (Solar Designer) in June 1997 for the Linux kernel [31], [36], [37] (Figure 2).

Just a few months after introducing the patch, Solar Designer himself described a novel attack that allows attackers to bypass a non-executable stack [38]. Rather than returning to code located on the stack, the exploit crafts a fake call stack mainly made of libraries' function addresses and arguments. Returning from the vulnerable function has the effect of diverting the execution to the library function. While any dynamically linked (and loaded) library can be the target of such

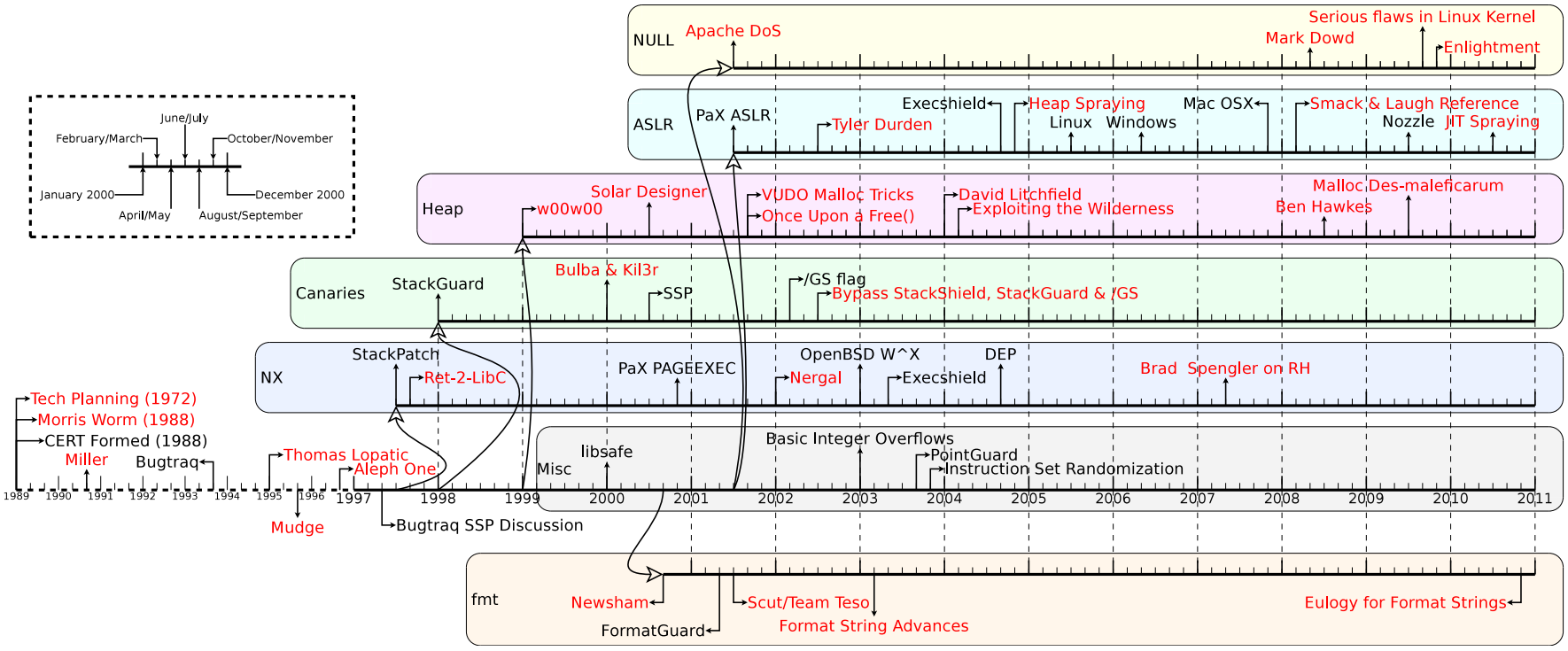

Fig. 1. General timeline

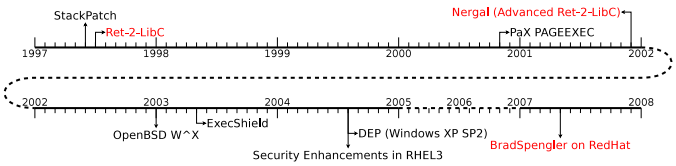

Fig. 2. Detailed Timeline of Non-eXecutable Stack.

diversion, the attack is often dubbed *return-into-libc* because the return address is typically replaced with C library functions with appropriate arguments (e.g., `system("/bin/sh")`).

A refinement to Solar Designer's non-executable stack patch was quickly proposed to withstand return-into-libc attacks [38]. However, shortly thereafter, Rafal Wojtczuk (Nergal) followed-up circumventing Solar Designer's refinement by taking advantage of specific ELF mechanisms (i.e., dynamic libraries, likely omnipresent functions, and dynamic libraries' function invocation via PLT, the ELF Procedure Linkage Table) [39].

McDonald [40] built on such results and proposed return-into-libc as a technique to act as a first stage loader to run the injected code in a non-executable segment. By returning to the `mprotect` system call on UNIX-like operating systems (OSes) or the `VirtualProtect` API on Windows-based OSes, attackers could explicitly set previously unmarked code-injected data regions as executable. This technique is commonly used to bypass *generic* non-executable data protection.

The PaX Team went far beyond a non-executable stack solution. With the PaX project released in the year 2000 [41], they offered a general protection against the execution of code injected in data segments. PaX prevents code execution on all data pages and adds additional measures to make return-into-libc much harder. Under PaX, data pages can be writable, but not executable, while code pages are marked executable but not writable. Most current processors have hardware support for the NX (non-executable) bit and if present, PaX will use it. In case the processor does not provide hardware support for making pages executable, PaX can emulate such support in software. In addition, PaX randomizes the `mmap` base so that both the process' stack and the first library to be loaded will be mmapped at a random location, effectively the first form of address space layout randomization (Section VII).

One of the first attacks on PaX' ASLR was published by Nergal [39] in December, 2001. He introduced advanced return-into-libc attacks and exposed several weaknesses of the `mmap` base randomization. He showed that it is easy to obtain the addresses of libraries and stacks from `/proc/[pid]/maps` for a local exploit. Moreover, if the attacker can provide the payload from I/O pipes rather then the environment or arguments, then the program is exploitable. The information about library and stack addresses can also leak due to format bugs in the program (Section **??**).

OpenBSD version 3.3, released in May 2003, featured various buffer overflow solutions [42], broadly divided in four categories. The first measure was to cleanse poorly written `pmap` modules and enforcing `PROT_EXEC` as an independent flag rather than an implied one when a user requests a page with `PROT_READ`. This worked for many architectures but not the popular i386 and PowerPC (because of the way they execute permissions on a per-page basis, only the stack could be made non-executable). As a next step OpenBSD enforced what it termed W^X (a term that has since found wide adoption): memory cannot be both writable and executable. As a third step it made `.rodata` segments accessible only with `PROT_READ` permissions (unlike earlier implementations that offered `PROT_READ|PROT_EXEC` permissions). By providing a separate and read-only `.rodata` segment, BSD prevented attackers from looking for data that look like instructions and executing those. Lastly, OpenBSD adopted ProPolice (Section IV).

By this time all major OSes were picking up on buffer overflow solutions. Red Hat introduced new security enahancements to combat buffer overflow attacks in its Enterprise Linux Version 3 [43]. It featured a new kernel-based security solution termed ExecShield [44]. Similar to PaX, ExecShield makes a large part of the virtual memory address space non-executable, rather than just the stack. ExecShield also randomizes various parts: stack, location of shared libraries, start of programs heap, and the text region, with position independent executables (PIE). Exec Shield also includes a version of ProPolice, known as Stack Smashing Protector (SSP) (see Section IV).

In August 2005, Microsoft released Service Pack 2 (SP2) of the Windows XP OS, which included Data Execution Protection (DEP)—which prevented code execution from a programs' memory [15]. Like PaX, DEP came in two flavors: an hardware-enforced DEP and a software-enforced one.

Non-executable stack was considered a strong protection against code-injection attacks and vendors soon backed up software implementations by hardware support for non-executable data. However, techniques like return-into-libc soon showed how non-executable memory can only partially mitigate memory errors from being exploited.

In 2005, Krahmer [45] was the first to focus on short code snippet reuse instead of entire libc functions for exploit functionality—a direction that reached its zenith in return-oriented programming. This had another advantage also: the original return-into-libc attacks worked well on x86 CPUs, but much less so on 64-bit architecture where function arguments are passed within registers. In return oriented programming (ROP), attackers chain Krahmer's snippets together to create *gadgets* that perform predetermined but arbitrary computations [19]. The chaining works by placing short sequences of data on the stack that drive the flow of the program whenever a return instruction executes.

Recently, researchers have proposed ROP-specific countermeasures, but they have not seen deployment in mainstream OSes yet. Conversely, low-overhead bounds checkers [5], [6] and practical taint-tracking [46] may be viable solutions to defeat control-hijacking attacks.

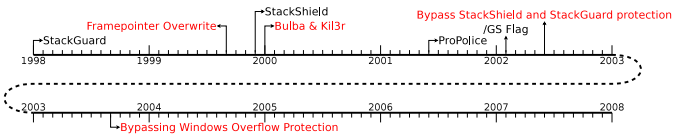

Fig. 3. Detailed Timeline of Canary-based Protections.

IV. Canary-based Protections

Canaries represent a first line of defense to hamper classic buffer overflow attacks. The idea is to use hard-to-predict patterns to guard control-flow data. The first such systems, StackGuard, was announced on December 18, 1997 [47] and released on January 29, 1999 [7]. When entering a function, StackGuard places a hard-to-predict pattern—the canary—adjacent to the function's return address on the stack. Upon function termination, it compares the pattern against a copy. Any discrepancies would likely be caused by buffer overflow attacks on the return address and lead to program termination.

StackGuard assumed that corruption of the return address only happens through direct buffer overflows. Unfortunately, indirect writes may allow one to corrupt a function return address while guaranteeing the integrity of the canary. StackShield [48], released later in 1999, tried to address this issue by focusing on the return address itself. Upon function entry, the return address is copied to a separate region of the address space, not reachable from the stack with a straight overflow. Upon function termination, the (safe) copy of the function's return address is checked against the actual function's return address. Any mismatch is likely caused by an overflow corrupting the return address on the stack and leads to program termination.

StackShield shows that in-band signaling should be avoided. Unfortunately, as we will see in the next Sections, mixing up user data and program control information is not confined to the stack: heap overflow (dynamic memory allocator metadata corruption) and format bug vulnerabilities intermixed (in-band) user and program control data in very similar ways.

Both StackGuard and StackShield, and their Windows counterparts, have been subject to a slew of evasions, showing how such defenses are of limited effect against skilled attackers [49], [50]. On Windows, David Litchfield introduced a novel approach to bypass canary-based protections by corrupting specific exception handling callback pointers (structured exception handing, SEH, exploits) used during the program cleanup phase, when return address corruption is detected [51].

Matt Miller subsequently proposed a solution to protect against SEH exploitation in 2006 [52] that was adopted by Microsoft (Windows Server 2008 and Windows Vista SP1). It organizes exception handlers in a linked list with a special and well-known terminator that is checked for validity when exceptions are raised. As SEH corruptions generally make such terminators unreachable, they are often easy to detect. Unlike alternative solutions introduced by Microsoft [53], [54], Miller's countermeasure is backward compatible with legacy applications. Besides, if used in conjunction with ASLR, it hampers the attackers' ability to successfully exploit SEH.

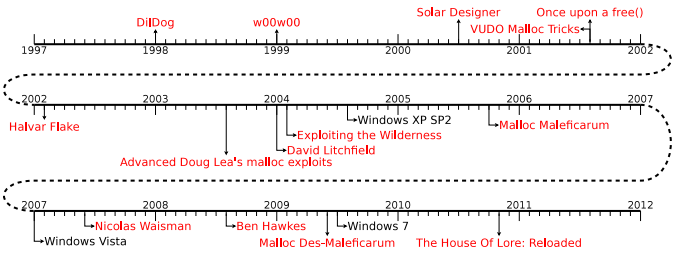

Fig. 4. Detailed Timeline of Heap Attacks.

Despite their initial weaknesses, canary-based protection spun off more counter measures. ProPolice, known also as Stack Smashing Protection (SSP), built on the initial concept of StackGuard, while addressing its shortcomings [55]. In particular, it rearranged the layout of variables on the stack to avoid pointer corruptions by buffer overflows. SSP was successfully implemented as a low-overhead patch for the GNU C compiler 3.x and was included in mainstream from version 4.1. FreeBSD, OpenBSD, DragonflyBSD, and Ubuntu all use Stack Smashing Protection as a standard protection against stack overflows.

V. Heap Attacks

When defense mechanisms against stack-based buffer overflow exploitations were deployed, heap-based memory errors were not taken into consideration yet (see Figure 4).

The first heap-based buffer overflow can be traced to January 1998 [56], while a paper published by the underground research community on heap-based vulnerabilities appeared a year later [32]. While this represented low hanging fruit (as more advanced heap-based exploitation techniques were yet to be disclosed), it nonetheless pointed out that memory errors were not confined to the stack.

The first description of more advanced heap-based memory error exploitations was reported by Solar Designer in July, 2000 [57]. The exploit again shows how in-band control information (heap management metadata) are a bad practice and should *always* be avoided, unless integrity checking mechanisms are in place. While a successful heap-based exploitation (heap management metadata corruption) is harder than its stack counterpart, the resulting attack is more powerful. In the end, the attacker obtains on a *write-anything-anywhere* primitive that allows him to eventually execute arbitrary code. Detailed, public disclosure of heap-based exploitations appeared in [58], [59]. Such papers dug into the intricacies of the System V and GNU C library implementations, providing the readers with all the information required to write robust heap-based memory error exploits.

Initially, limited to UNIX environments, Windows OSes were not immune from heap exploitation either. BlackHat 2002 hosted a presentation by Halvar Flake on the subject [60], while more advanced UNIX-based heap exploitation techniques where published in August 2003 [61]. It describes how to obtain write-anything-anywhere primitive and information

leaks, that an attacker can use to exploit a system, even when ASLR is in use.

More about Windows-based heap exploitations followed in 2004 [62]. With the introduction of Windows XP SP2, later that year (August), the heap became non-executable. In addition, SP2 introduced heap cookies, canary-like protections, and safe heap management metadata unlinking (whose unsafe version was responsible for the write-anything-anywhere primitive). Before long, however, the first working exploits against Microsoft latest updates appeared [63], [64], [65], [66], [67], [68]. The heap exploitation story continues in 2004 with even more esoteric exploitation techniques [69], which worked even against dynamic memory allocator enhanced with integrity checks placed to detect in-band management data corruptions.

During these frentic years, attackers kept jumping from UNIX systems to Windows-based OSes at will. With the release of Windows Vista in January 2007, Microsoft further hardened the heap against exploitation [70]. However, as with the UNIX counterpart, there were situations in which application-specific attacks against the heap could still be executed [71], [72].

In 2009 and 2010 a report appeared where proof of concept implementations of almost every scenario described in [73] were shown in detail [74], [75].. [75] also discusses the effectiveness of ASLR and a non-executable heap as a defense mechanism against heap-based exploitations. As expected, non-executable heaps can be defeated with return-into-libc-based approaches. Conversely, novel techniques, such as heap spraying and heap Feng Shui (see Section VII), were introduced to withstand ASLR.

Over time, we can identify three different generations in heap exploitations: (1) classic overflows to corrupt adjacent memory locations, (2) heap management metadata corruptions providing a write-anything-anywhere primitive that allows for execution of arbitrary code, and (3) heap spraying and application-specific attacks to bypass heap protection mechanisms. Both the first and second generation of attacks have been mitigated by hardening dynamic memory allocators. Examples include the introduction of heap cookies, safe heap management metadata unlinking, and the addition of consistency checks during allocation and deallocation of heap.

Apart from heap-specific mitigation approaches, dynamic memory allocators also benefit from alternative defenses against memory error—e.g., non-executable heap and heap randomization enabled together to detect arbitrary code execution on the heap. Unfortunately, as shown in Section III, NX protections can be bypassed by return-into-libc or return-oriented programming attacks. Similarly, exploiting heap-based memory errors on a randomized heap is nowadays mostly achieved with heap spraying, a third generation of heap attacks (see Section VII).

We conclude that, although some special cases allowing for successful heap exploitations, heap allocators have become resilient against most attacks. Application-specific heap exploits are still sometimes possible [74], [75], but in the end, allocator implementations have become stable enough to prevent easy heap exploits. Research and industry efforts should probably focus on providing both *effective* and *efficient* protection against heap spraying attacks, as they are responsible for a large portion of malware infections [18].

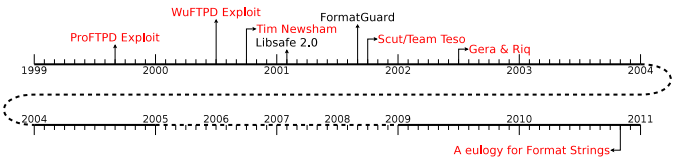

Fig. 5. Detailed Timeline of Format String Attacks.

VI. FORMAT STRING ATTACKS

Format string vulnerabilities affect the `printf` family of functions. These variable arguments functions take usually a *format string* as an argument and a series of additional arguments, accordingly to the formatting string. If the format string is under the control of an attacker (e.g., `printf(buf)`), the vulnerability can be exploited. Depending on the formatting directive used, double words can be directly (e.g., `%x`) or indirectly (e.g., `%s`) retrieved from the vulnerable process address space. Moreover, the number of bytes written so far by such functions can also be written at the next address to be retrieved from the stack (typically), by using the `%n` or one of its variants (e.g., `%hn`, `%hhn`, `%`k`$n`).

Similarly to the second generation of heap attacks, but unlike classic buffer overflows, format string vulnerabilities are easily exploited as a write-anything-anywhere primitive, potentially corrupting the whole address space of a victim process. Besides, format bugs also allow to perform *arbitrary* read of the whole process address space. Disclosing confidential data (e.g., cryptographic keys and seeds used by some memory error countermeasures [7], [10]), executing arbitrary code, and exploring the whole address space content of a victim process are all viable possibility.

Format string vulnerabilities were first discovered in 1999 while auditing ProFTPD [33], but it was in the next couple of years that format strings gained much popularity. A format string vulnerability against WU-Ftpd was disclosed on Bugtraq in June 2000 [76], while Tim Newsham was the first to dissect the intricacies of the attack, describing the fundamental concepts along with various implications of having such vulnerability in your code.

One of the most extensive articles on format string vulnerabilities was published by Scut of the TESO Team in September 2001 [77]. Along with detailing conventional format string exploits he also presented novel hacks to exploit the vulnerability. Response-based brute force attacks, which take advantage of the format reply output, and blind brute force attacks allowed to overcome some of the complex bits of the attack. Besides, by having a write-anything-anywhere primitive at hand, Scut showed that it was easy to target the corruption of alternative control-flow data (e.g., global offset table entries, dtors). Not only such targets where stored at well-known locations in memory, but they also allowed

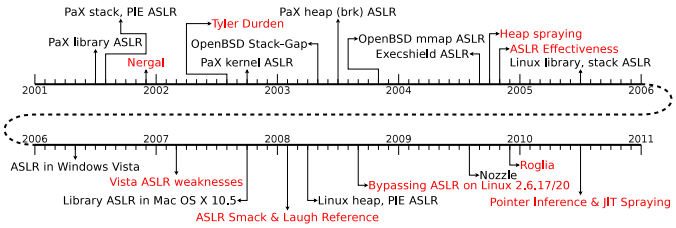

Fig. 6. Detailed Timeline of ASLR Approaches.

to bypass some basic stack-based protections (e.g., canaries, return address integrity checks) [78].

Protection against format string attacks were proposed in [79]. FormatGuard, the codename of the approach, uses static analysis to compare the number of arguments supplied to `printf`-like functions with those actually specified by the function's format string. Any mismatch would then be considered as an attack and the process terminated. Unfortunately, the effectiveness of FormatGuard is bound to the limits of static analysis, which leaves exploitable loopholes.

Luckily, format string vulnerabilities are generally quite easy to spot and the fix is often trivial. Moreover, since 2010, the Windows CRT disables `%n` directives by default [80]. Similarly, the GNU C library `FORTIFY_SOURCE` patches provide protection mechanisms, which make format string exploitations hard. Although the low hanging fruit had been harvested long ago, the challenge of breaking protection schemes remains still exciting [81].

VII. Address Space Layout Randomization

Memory error exploitations usually require an intimate knowledge of the vulnerable process address space to succeed. In particular, attackers must often find suitable addresses to corrupt or to divert the execution to. Consequently, any attempt to randomize the memory locations of such objects would increase the likelihood of resisting to memory error attacks.

The PaX Team proposed the first form of address space layout randomization (ASLR) in 2001 [41]. ASLR can be summarized succinctly as to introduce randomness in the address space layout of userspace processes. Such a randomness would make a class of exploits fail with a quantifiable probability and would also allow their detection as failed attempts will most likely crash the vulnerable process.

This section details how ASLR works and what ASLR-related events occurred after its first introduction (Figure 6).

PaX-designed ASLR underwent many improvements over the time. The year 2001 was definitely the most prolific for the team. The first PaX-devised ASLR implementation provided support for `mmap` base randomization (July). When randomized `mmap` base is enabled, dynamically linked code (e.g., shared objects) are mapped at a different, randomly selected offset each time a program starts. This causes dynamically-linked library functions to be located at different addresses, which makes makes return-int-libc attacks difficult. Stack-based randomization followed quickly in August 2011. While code injection per-se is not avoided by stack-based randomization, finding the injected code becomes hard. Position-independent executable (PIE) randomization were proposed in the same month. PIE binaries are similar in spirit to dynamic shared objects. That allows to load PIE binaries at arbitrary addresses, which reduces the risk of performing successful return-into-plt or more generic return-oriented programming attacks. As stack-based buffer overflows and code injection attacks in the kernel were becoming popular, the PaX Team proposed a kernel stack randomization in October 2002. Finally, similar to the way the `mmap` base address was randomized to avoid return-into-libc attacks, the PaX Team released a patch to randomize the heap of processes.

The PaX project was first released as a series of patches for the Linux kernel, but it was OpenBSD the first to include such concepts in its mainstream kernel [14]. However, the OpenBSD team independently conceived its own exploit mitigation techniques, including simple stack randomization [82], stack-smashing protection (SSP), non-executable memory, dubbed as W^X [42] and `mmap` base address randomization [83]. However, OpenBSD did not support kernel stack randomization, as that would have broken the POSIX standards. To this end, an intense debate between de Raadt and the PaX Team about who first proposed randomization-based protections, non-executable memory and whether standards were broken or not, was started shortly thereafter in April 2003 [84], [85], [86], [87].

Red Hat ExecShield added support for ASLR in August, 2004 [43]. That latest version supports stack, `mmap` base address and heap randomization. It also supports PIE binaries, providing a comprehensive kernel-enforced randomization.

The Linux kernel enabled stack and `mmap` base address randomization by default since their 2.6.12-rc1 kernel, released on March 2005 [88]. The first patches for randomization were released on January 2005 [89], by Arjan van der Ven, who, at that time, was also working on the Red Hat ExecShield project. It took until April 2008 to Linux to include heap and PIE randomization [90]. Although it lacks kernel stack randomization, Linux supports *full* ASLR since 2008 [91].

Microsoft added ASLR support to their new Windows operating system as well. The first Windows version with stack, heap and library randomization was Windows Vista Beta 2, released on May 26, 2006 [92]. Shortly after this release, Ali Rahbar stated in his analysis of Microsoft Windows Vista ASLR that the implementation suffered from the presence of bugs [93]. In a response, Howard refuted these accusations [94]. Later, when Windows Vista was officially released, another analysis of ASLR on Windows Vista was done by Whitehouse [95]. He concludes that "the protection offered by ASLR under Windows Vista may not be as robust as expected". It is uncertain what happened after this when Windows 7 was released. Although the deficiencies in the ASLR implementation have been acknowledged by Microsoft, there is little to find about any follow up. In June, 2010, however, Alin Rad Pop published a paper discussing the use of ASLR and DEP in third-party Windows applications. He concludes

that third-party are quite slow in adding ASLR support to their applications. In June 2010, only Google Chrome and Adobe Flash Player were using full ASLR protection. Popular applications like Adobe Reader, Mozilla Firefox and Apple iTunes did not have full ASLR support when being executed on the Windows platform [96].

Apple introduced partial ASLR support in the Mac OS X 10.5, dubbed as library randomization [97]. Although stack and heap protections are supported via non-executable data, as its name suggests, only library functions are actually randomized, and full ASLR is not supported yet.

Broadly speaking, only coarse-grained—often kernel-enforced—forms of ASLR were actually deployed. Such randomization techniques are generally able to randomize the *base* address of specific regions of a process address space (e.g., stack, heap, `mmap` area). That is, only absolute addresses are randomized, while relative offsets (e.g., the location of any two objects) in the library is fixed. An attacker is just left with retrieving the absolute address of a *generic* object of the library of interest: any other object (e.g., library functions used in return-into-libc attacks) can be reached as an offset from it.

To overcome such limitations, Bhatkar et al. proposed *fine-grained* address space randomization (ASR) approaches to allow for arbitrary fine-grained objects randomization [98], [11]. They first propose a randomization scheme that, through binary-rewriting techniques, is able to obfuscate the stack, `mmap` and heap base addresses, and the code section layout of legacy programs. Because it relies on specific information to be present in the binary, the approach effectiveness is bound to the accuracy in which such information are available (e.g., function randomization relies on the ability to detect function boundaries) [98]. Conversely, the second approach describes a source-to-source transformation technique that produces self-randomizing PIE-like binaries.

Heap spraying attacks (described in the next section) were instead addressed in August 2009 by Nozzle [99], which monitors heap utilization and raise an alarm when a high fraction of the heap region contains suspicious objects. Earlier, in July 2009, Egele et al. proposed a similar technique that uses emulation to identify JavaScript strings likely representing shellcode. By integrating this detector in the browser, they could successfully detect thousands of infected websites used to carry out drive-by-download attacks [18].

A. Attacking ASLR

One of the first attacks against ASLR was presented by Nergal in 2001 [39]. Although the paper mainly focuses on bypassing non-executable data protections, the second part addresses PaX randomization. Nergal describes a novel technique, dubbed return-into-plt, that enables to call directly the dynamic linker's symbol resolution procedure, which is used to obtain the address of the symbol of interest. Such an attack was however defeated when PaX released PIE.

In 2002, Tyler Durden showed that certain buffer overflow vulnerabilities could be converted into format string bugs, which could then be used to leak information about the address space of the vulnerable process [100]. Such information leaks would become the de-facto standard for attacks on ASLR.

In 2004, Shacham et al. showed that ASLR implementations on 32-bit platforms were of limited effectiveness. Due to architectural constraints, and kernel design decisions and modus operandi the available entropy is generally limited and leaves brute forcing attacks as a viable alternative to exploit ASLR-protected systems [101].

Tilo Müller provides an in-depth discussion about attacks against ASLR in [102]. He mentions a number of variations to the popular return-into-libc (e.g., return-into-text, return-int-bss, return-into-data, return-into-return) and many other attack types. Some of these techniques, such as return-into-text or return-into-got, could indeed be useful to bypass non-executable data protections as well. He concludes that "ASLR and, therefore, e.g., a standard Linux installation, is still highly vulnerable against memory manipulation." Note that Linux implemented heap-based and PIE-based ASLR only two months after Müller's research.

FHM crew and others explored the possibility to use specific instruction of non-randomized shared libraries. The linux gate shared library is a virtual dynamically-linked shared object (VDSO) that bridges user and kernelspace interactions (i.g., system call invocations) by using fast instructions (e.g., `sysenter` and `sysexit`), if available on the considered architecture. Early Linux kernel did not randomize the virtual address of the linux gate VDSO and that was thus used in return-into-lib-like attacks [103].

Finally, Fresi-Roglia et al. [104] detail a return-oriented programming [19] attack able to bypass WˆX and ASLR. Such an attack chains code snippet of the original executables and, by copying data from the global offset table, is then able to compute the base addresses of dynamically linked shared libraries. Such addresses are later used to build classic return-into-lib attacks. The attack proposed is estimated to be feasible on 95.6% binaries for Intel x86 architectures (61.8% for x86-64 architectures). This high success rate is caused by the fact that modern OSes do not adopt or lack PIE (Fresi-Roglia et al. propose a solution, which is low-overhead, does not require recompilation, and represents a valid alternative to PIE)

A different class of attacks against ASLR protection, called heap spraying, was described first in October 2004 when SkyLined published a number of heap spraying attacks against Internet Explorer [105], [106], [107]. By populating the heap with a large number of objects containing attacker-provided code, it is possible to increase the likelihood of success in referencing (and executing) such code.

Heap spraying is mostly used to exploit cross-platform browser vulnerabilities. Since scripting languages like JavaScript and ActionScript are executed on the client's machine (typically in web browser clients), heap spraying has become the main infection vector of end-user hosts. The technique has been improved in March 2007 [108], where a novel and reliable technique for precise manipulation of the browser heap layout using specific sequences of JavaScript allocations is presented.

Dion Blazakis went far beyond heap spraying by describing pointer inference and JIT spraying techniques [109]. He admits that "[...] these techniques leverage the attack surface exposed by the advanced script interpreters or virtual machines commonly accessible within the browser. The first technique, pointer inference, is used to find the memory address of a string of shellcode within the ActionScript interpreter despite ASLR. The second technique, JIT spraying, is used to write shellcode to executable memory by leveraging predictable behaviors of the ActionScript JIT compiler bypassing DEP".

Wei et al. followed-up and proposed dynamic code generation (DCG) spraying, a generalized and improved JIT spraying technique [110]. (Un)luckily DCG suffers from the fact that memory pages, which are about to contain dynamically-generated code, have to marked as being writable *and* executable. However, Wei et al. found that all DCG implementations (i.e., Java, Javascript, Flash, .Net, Silverlight) are vulnerable against DCG spraying attacks. A new defense mechanism to withstand such attacks were eventually proposed [110].

Finally, return-oriented programming, introduced in Section III, may also be used to bypass non-PIE ASLR-protected binaries (as shown by [104]. In fact, for large binaries, the likelihood of finding enough useful code snippets to build a practical attack is non-negligible [111].

B. ASLR Effectiveness

Despite the persistent arms-race, address space (layout) randomization techniques have shown their effectiveness in providing protection against a broad class of memory errors, not limiting themselves to just buffer overflows.

The underlying idea is inspired by nature, where diversity plays a fundamental role for the survivability of the species. Roughly speaking, the key observation here is that memory error exploitations generally rely on finding suitable memory addresses to be used as part of the attack. As a consequence, forms of process address space diversification, such as ASLR, generally aim at randomizing such memory addresses to make them unpredictable to an attacker.

Unfortunately, the main drawback of ASLR is its *probabilistic* nature. By relying on keeping secrets, ASLR is vulnerable to information leakage. Likewise, architectural constraints limit the degree of address space randomization and, finally, common forms of ASLR randomize only the base addresses of a process memory segments, leaving exploitable loopholes.

Ever since its first implementation in mid 2001, a number of ASLR evasions techniques were discussed by both the academia and underground community. While the first attacks had focused on targeting memory regions not protected by randomization yet, the remaining could be divided in two categories: brute force [101] and information leakage [104], [112].

Return-oriented programming attacks could be mitigated by moving to 64-bit architectures and full ASLR, including the support of PIE binaries (given that no other information about the vulnerable process address space leak). Unfortunately, very few binaries are found to be PIE, to date, which leaves return-oriented programming techniques a viable vector to exploit non-executable and ASLR-protected systems [104].

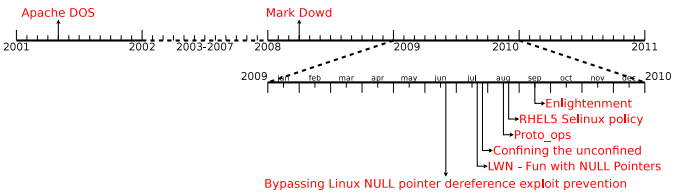

Fig. 7. Detailed timeline for NULL Pointer Dereferences

Other diversity-inspired *deterministic* solutions have recently been explored by the research community [113], [114], [115]. Such approaches combine diversification with replication and, depending on the underlying technique, they are able to deterministically withstand code and data pointer corruptions. Unfortunately, the high overhead imposed along with subtle behavior inconsistencies that may arise, leave such approaches confined to research environments.

VIII. NULL Pointer Dereference

`NULL` pointers are, by convention, pointers that do not have any actual referent. They are generally used to represent specific conditions, e.g., lists termination and string terminations, in low-level programming languages, like C. Besides, OSes typically do not map the first page of a process virtual address space to catch `NULL` pointer dereference attempts. In fact, `NULL` pointer dereference vulnerabilities are generally extremely difficult—if not impossible—to exploit. Barnaby Jack presented some research on `NULL` pointer dereference exploitation for specific architectures [116], while Matt Miller (skape) and Ken Johnson (Skywing) explored such exploitations on the Windows operating systems [117].

It is Mark Dowd, however, that presented a top-notch research to exploit a `NULL` pointer vulnerability against the Adobe Flash Player [118]. By leveraging functionality provided by the ActionScript virtual machine, Mark is the first to point out that such vulnerabilities will unlikely be exploitable by conventional-only techniques. Conversely, his findings show that classic attack techniques *combined* with application-specific attacks are successful in producing reliable likely cross-platform exploits [35].

A large number of such vulnerabilities have been reported more recentlyIn June, 2009 Julien Tinnes and Tavis Ormandy [119] described a technique to evade a check performed by the Linux Security Module (LSM) hooks subsystem. The relevant code in `security/capability.c` prevents VM pages below `mmap_min_addr` to be memory mapped (to hamper classic `NULL` pointer dereference attacks). Unfortunately, processes with specific capabilities (i.e., `CAP_SYS_RAWIO`) could bypass this security check.

Similarly, Brad Spengler described a Red Hat Linux-based exploit that allowed specific SELinux domains to map the zeroth page [120]. The exploit is interesting in many aspects, but first and foremost, because it points out how compiler

optimizations may, once again, be responsible of WYSINWYX events [121] and open unexpected loopholes [122]. Patches were provided, but ways to bypass them still existed, as reported by Dan Walsh [123].

Finally, August 2009 saw Julien Tinnes and Tavis Ormandy once again leading actors describing another `NULL` pointer vulnerability concerning the way the Linux kernel deals with unavailable operations for some network protocols [124].

Undoubtedly, security practitioners termed 2009 as the year of kernel `NULL` pointer dereference [125].

IX. Alternative Defenses (and Attacks)

In this section, we quickly introduce a number of memory error-related vulnerabilities, attacks, and defenses that do not easily blend into the discussion faced earlier.

To start with, `libsafe` was one among the first buffer overflow prevention mechanisms proposed in 2001 [126]. The underlying idea is to determine upper bounds on the buffers size automatically. The assumption made by `libsafe` is simple: local buffers should never extend beyond the end of the current stack frame. Of course, such a computation can only be made at run-time, right after the execution of the function in which the buffer is accessed starts. The library is dynamically loaded and potentially vulnerable library functions are replaced by `libsafe`-provided ones. Although `libsafe` does not offer a comprehensive solution to memory errors, it detects return address corruptions, retrofits existing binaries, and has a low-overhead impact on the performance.

A successful memory error exploitation usually requires to corrupt code or data pointers of interest[1]. PointGuard aims at protecting all pointers from corruptions [128]. The underlying idea focuses on encrypting pointer values in memory and decrypting them right before use by the CPU. Unfortunately, PointGuard is vulnerable to information leak (via format string) and partial overwrite attacks [22].

Code-injection attacks aim to exploit memory error vulnerabilities to hijack a process execution flow to the attacker-injected code. Instruction set randomization (ISR) is a technique aimed at withstanding code injection attacks [13], [12]. Similar in principle to PointGuard, ISR-based approaches encrypt a program instructions with randomly a per-process randomly generated key and decrypt them right before execution by the CPU. As we have discussed earlier, other attack vectors (e.g., return-into-libc or the more generalized return-oriented programming) can easily bypass such defenses.

A. Integer Vulnerabilities

Integer overflows are not memory errors by themselves [129]. However, incorrect integer handling can trigger memory errors, such as buffer overflows or write-anything-anywhere-like primitive, depending on the involved integer misinterpretation. The issue arises because of integer representations on computers. For instance, on IA-32 an `unsigned int` type is usually 4 bytes wide, while 2 bytes are needed for an `unsigned short int` type. If the value assigned to an `unsigned short int` variable is $2^{16} - 1 - k$, that is far from its maximum value of k, adding $k + 1$ will cause the variable to wrap around reaching 0 (a similar reasoning can be made for underflow). This can be used to bypass security checks or write to arbitrary memory regions, especially when `unsigned int` variables are involved[2].

A more subtle way to exploit integer overflows is caused by the fact that two different representations are used depending whether the considered integer is *unsigned* or *signed*. If improperly considered, such an idiosyncrasy could easily lead to buffer overflows even when bounds check conditions where enforced in the first place [6].

Integer overflows were first discussed in the public during Black Hat 2002 [130]. In the same year, Phrack published two articles on integer vulnerabilities as well. The first one focused on basic integer overflows and discussed different types of integer vulnerability and how they could exploited [129]. Conversely, the second article proposes a compiler-enforced protection mechanism against integer overflows [131].

Over time, several other attempts of protection against integer vulnerabilities have been researched. They range from static type checkers [132] and symbolic execution-based approaches [133], to safe integer arithmetic libraries, which provide safe memory allocation and array offset computations for C [134] and C++ programs [135], [136], [137], but leave all the burden on the programmer's shoulders.

Limited integer overflow defense was also introduced in the Linux kernel by Brad Spengler [138], but, despite all such efforts, integer errors are still being leveraged to exploit the memory errors such vulnerabilities expose.

X. Data Analysis

The previous Sections aimed at providing an as comprehensive as possible view of the most important events and facts about memory errors. This Section aims at analyzing real-life evidence as well as statistics about vulnerability and exploit reports to draw a final speculative conclusion about memory errors: are we going to be living with them for a while, or are memory errors an heritage of the past?

We looked up occurrences of vulnerabilities and exploits over the past 15 years by examining the Common Vulnerabilities and Exposures (CVE) and ExploitDB databases.

Figure 8 shows that memory error vulnerabilities have grown almost linearly between 1998 and 2007 and that they started to attract attackers in 2003, where we witness a linear growth in the number of memory error exploits as well. The downward trend in discovered vulnerabilities that started in 2007 is remarkable. Instead of a linear growth, it seems that the number of found vulnerabilities is now reversed. To understand the reasons, Figure 9 reports the total number of vulnerabilities as well as the data depicted by Figure 8.

[1] It has been shown, however, that non-control data attacks are as powerful and dangerous as their control-hijacking counterparts [127].

[2] In fact, `unsigned int` variables are 32 bits wide and so they can be used to address the whole default user space process address space on IA-32 machines.

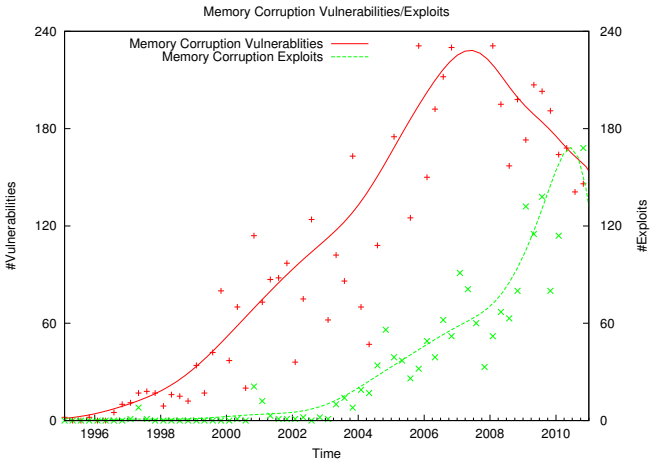

Fig. 8. Memory Error Vulnerabilities and Exploits.

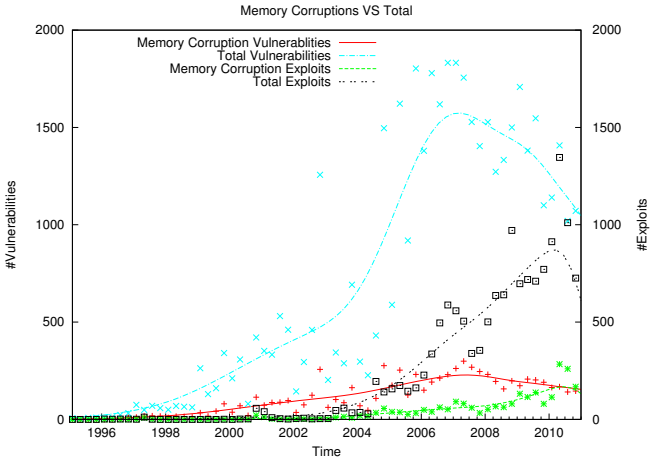

Fig. 9. Memory Error Vulnerabilities and Exploits compared to Totals.

Figure 9 shows that the drop in memory error vulnerability *reports* is probably caused by a similar drop in the total number of vulnerability reported. We will try to clarify this downward trend shortly, but first let us have a look at the reasons that may have caused the immense growth of vulnerabilities started in 2003.

A. The Rise...

Interestingly, the spike in the number of vulnerabilities started in 2003 might have been caused by the number of *web* vulnerabilities that popped up during that period. An independent research by Christey and Martin in 2007 seem to support such a claim [139]. Figure 10 augments the previous with web vulnerabilities (i.e., XSS, SQL injection, and PHP-related) and the corresponding exploits.

Figure 10 points out that web vulnerabilities showed up in 2003 and indeed outgrew the number of buffer overflow vulnerabilities rapidly. Probably due to its simplicity, the number of working web exploits also transcended the number of buffer overflow exploits in 2005.

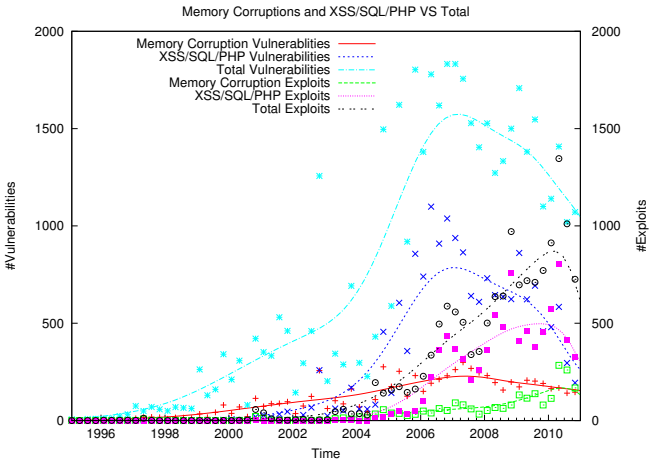

Fig. 10. Memory Error, XSS, SQL, and PHP Vulnerabilities and Exploits compared to Totals.

It seems therefore obvious to conclude that the extreme growth in vulnerability reports that started in 2003 was caused by the fast growing number of web vulnerabilities. Shortly after the dot-com bubble in 2001, when the web 2.0 started to kick-in, novel web developing technique were not adequately tested against possible exploitation techniques. This is probably due to the high rate at which new features were constantly asked by end customers: applications had to be deployed quickly in order to keep up with competitors. This race apparently left no time to developers to more carefully perform security audits of the code produced.

B. ...And the Fall

Figures 9 and 10 show a similar trend in the total number of vulnerabilities over the years 2006-2010, as reported independently in [21] as well. Memory errors were also affected by that drop and they start a downward trend in early 2007, indeed. Despite these drops, generic exploits and memory error-specific exploits kept growing linearly each month.

The reasons of such downward state could be manifold. For instance, it could be that less bugs are found in source code, less bugs are reported, or, instead, a combination thereof.

Assuming that the software industry is still growing and that hence the number of lines of code (LoC) written each month still increases, it is hard to back the first statement up. More LoC naturally results in more bugs: software reliability studies have shown that executable code may contain up to 75 bugs per 1000 LoC [140], [141]. CVEs look at vulnerabilities and does not generally make a difference between plain vulnerabilities and vulnerabilities that could be exploited. Therefore, memory error mitigation techniques could not have contributed to the drop in (reported) vulnerabilities. Most defense mechanisms that we have discussed earlier do not result in safer LoC; they only prevent exploitation of poorly written code.

However, if we look more carefully at the data provided by analyzing CVE entries, as depicted in Figure 10, we see that the number of web vulnerabilities follows the same trend

as that of the total number of vulnerabilities. Hence, we think that both the exponential growth (2003–2007) and drop (2007–2010) in vulnerabilities is correlated to fundamental changes in web development. We believe that companies, and especially their web developers, started to take web programming more seriously in 2007. For one thing, developers probably became more aware of how easy things like SQL injections or XSS could be accomplished, which may have raised web security concerns, resulting in better code written.

The year 2007 could also have been the period when developers switched from their home-made content management systems, managed and hacked together by a single person, to full-fledged `PHP` frameworks (e.g., `Joomla!`, `Zend`), which are updated on a regular basis. Such a switch would also result in a drop of vulnerabilities being reported. Something similar could have happened to servers codebase (e.g., `Apache`). For instance, if at some point `Apache`'s default `PHP` interpreter underwent a design change, the number of `Apache`-*reported*—but not necessarily related to its core codebase—vulnerabilities could have dropped significantly.

To substantiate the second statement (i.e., less bugs are reported), we need to have a more social view on the matter. There could be a number of reasons why people stopped reporting bugs to the community.

A first reason could be dubbed "The Great Recession". In the years before 2007, security experts were getting paid to look for vulnerabilities. Things changed when companies ran out of money: bug hunters were fired or placed in a different position to do some "real" work instead, resulting in less people searching for vulnerabilities.

A second reason could advocate for a "no full disclosure due to bounties". Ten years ago, the discovery of a zero-day vulnerability would have likely led to a patch, first, and a correspondence with the application authors/vendor about the fix, possibly via a public mailing list. Today, large companies, like Google and Mozilla, give out rewards to bug hunters, as long as they do not go public with the vulnerability. There is now real money to be paid for zero-day vulnerabilities.

In contrast, a third explanation could be rooted in having "less fun". Developers who used to spend their spare time on finding bugs and hacking into programs, have lost the fun part of doing this. On the one hand, programs and their software companies are becoming more professional over time. They do not like public disclosure about vulnerabilities concerning their software anymore: it makes them look bad and they could lose clients with it. On the other hand, new mitigation techniques may have made it harder for those spare time hackers to look for bugs. They could possibly find a vulnerability, but exploiting it and writing a proof-of-concept would take considerable more time than it used to.

Finally, the "criminal world" may instead be responsible for such a downward trend. While more and more people start buying things online and use online banking systems, it becomes increasingly more interesting for criminals to move their activities to the Internet as well. Where companies send out rewards to finders of vulnerabilities, useful zero-days in the underground market would yield even more. Chances that issues found by criminals are reported as CVE are negligible.

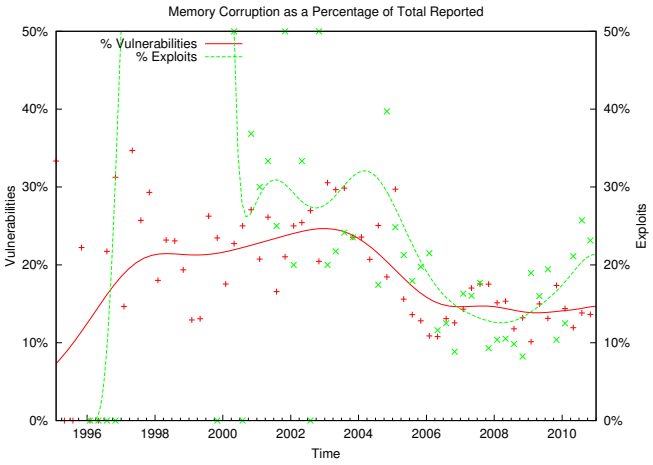

Fig. 11. Memory Error Vulnerabilities and Exploits (% of Totals).

We believe that the drop in vulnerabilities is caused by both previous statements. The software industry has become more mature during the last decade, which led to more awareness about what potential damage a vulnerability could cause. Web developers or their audits switched to more professional platforms instead of their home-brew frameworks and eliminated easy vulnerabilities by simply writing better code. This professionalization of the software industry also contributed to the fact that bugs are no longer reported to the public, but being sold to either the program's owners or the criminal underground. Full Disclosure [142] as it was meant to be, is being avoided. As an example for this shift in behavior, researchers got threatened for finding a vulnerability [143]. This was also recently backed up by Lemos and a 2010-survey that has looked at the relative trustworthiness and responsiveness of various organizations that buy vulnerabilities [144], [145].

C. Effectiveness of Deployed Mitigation Techniques

To help us better defining a reasonable final answer on the matter, we plot in Figure 11 memory error vulnerabilities and exploits as a percentage on the total numbers reported.

Figure 11 shows the same trend in percentage as that identified in [21]: memory error-related bugs were a very hot topic during the years 1996–2004, when more than 20 out of every 100 bugs were related to memory errors.

To clarify the drop in percentage of memory error vulnerabilities and exploits started in 2004, Figure 11 was augmented with web vulnerabilities and exploits, as shown in Figure 12.

Figure 12 once again shows that the focus of the security community definitely shifted towards the web when XSS, SQL injections and PHP-related issues became popular. It seems fair to conclude that the downward trend of memory error vulnerabilities is indeed caused by the upward trend of XSS, SQL, and PHP-related vulnerabilities. The same reasoning applies to exploits: the usually low technical skill required

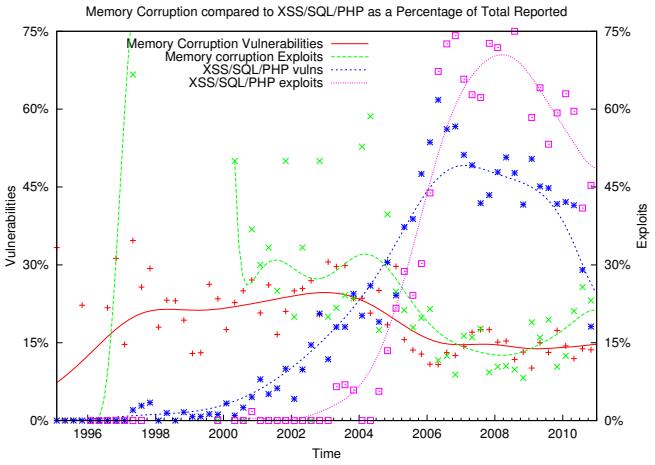

Fig. 12. Memory Errors Compared to XSS and SQL Injections (% of Totals).

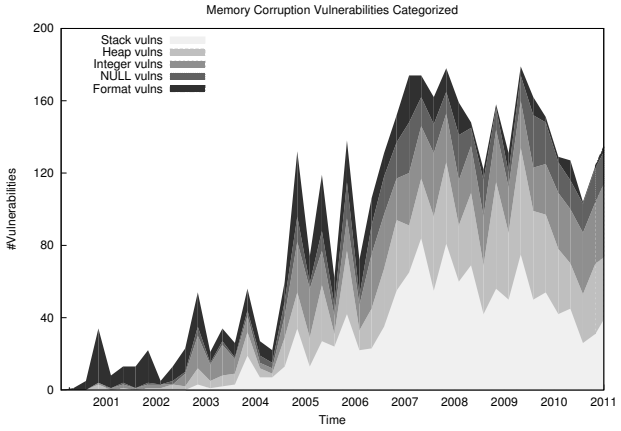

Fig. 13. Memory Error Vulnerabilities Categorized.

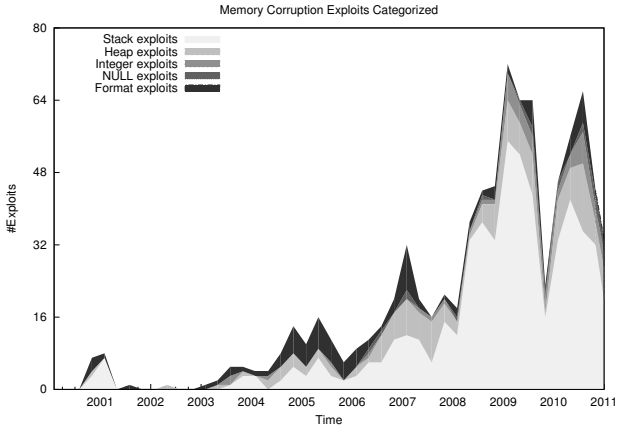

Fig. 14. Memory Error Exploits Categorized.

to exploit XSS and SQL issues spiked the number of web exploits to be almost 3x higher than memory errors.

However, both web vulnerabilities and exploits seem to have become less active over the last two years. This is by far in clear contrast with the percentage of memory error vulnerabilities and exploits. These numbers are steady since 2007 and memory error exploits may even increase in popularity in the near future considering the growth over the first 6 months of 2010. Not to mention that, besides other popular vectors (e.g., SPAM and phishing-like attacks), malware infections are triggered by drive-by-download attacks exploiting memory errors, even on protected systems [146], [147], [148].

We conclude that memory errors are still a security issue undermining the safety of our systems. Besides, it is not likely that they will be vanishing in the next years. It is actually hard to reason whether mitigation techniques affect the number of memory error exploits or not. Although the numbers would probably be worse if none of the such techniques were deployed, memory error exploits are still alive and contribute to a significant part of all the exploits. Attackers circumvent defense mechanisms by applying different techniques, or by simply exploiting bugs on systems that are not fully protected yet. Even worse, evidence shows that state-of-the-art detection techniques fail to protect such vulnerabilities from being exploited by well-motivated attackers [146], [147], [148].

D. Categorizing Vulnerabilities and Exploits

We furthermore categorized memory error vulnerabilities and exploits in 6 different classes (based on their CVEs descriptions): stack-based heap-based, integer issue, `NULL` pointer dereference, format string, and other (for whatever does not fit in the previous categories). Figures 13 and 14 show such classification for vulnerabilities and exploits, respectively (the "other" class was left out to avoid noise in the plots).

Figures 13 and 14 allow us to make the following observations, which may draw a final conclusion. First, format string vulnerabilities were found all over the place shortly after they were first discovered. Over the years, however, the number of format string errors dropped to almost zero and it seems that they are about to get eliminated totally in the near future. Second, integer vulnerabilities were booming in late 2002, and, despite a small drop in 2006, they are still out there right now. Third, despite the large number of `NULL` pointer dereference that were found over time, such vulnerabilities do not get exploited very often. This is probably because exploiting such vulnerabilities often require application-specific attacks that are not worth the effort, especially nowadays that memory errors are more often exploited by cyber-criminals. Last, the old-fashioned *stack* and *heap* memory errors are by far (about 90%) still *the most exploited* ones, while they counts just for 50% of all the reported vulnerabilities: there is no evidence that makes us believe this will change in the near future.

XI. Conclusion

Despite more than twenty years of research on software safety, memory errors are still one of the primary threats to the security of our systems. Not only is this confirmed by statistics, trends [149], [150], and our study, but it is also supported by evidence showing that even state-of-the-art *detection* and *containment* techniques fail to protect such vulnerabilities

from being exploited by motivated attackers [146], [147], [148]. Besides, protecting mobile applications from memory errors may even be more challenging [151].

Finding alternative mitigation techniques is not an academic exercise anymore, but a concrete need of industry and society at large: vendors have recently announced consistent cash prizes to researchers who will concretely improve on the state-of-the-art detection and mitigation techniques against memory error exploitation attacks [152].

However, given the trends in memory error exploitation, our tentative conclusion is that a mindset is needed where we assume that code can and will be exploited. In addition to our efforts on preventing memory corruption, this suggests that we should pay (even) more attention to the containment of such attacks.

REFERENCES

[1] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *USENIX ATC*, 2002.

[2] G. C. Necula, J. Condit, M. Harren, S. Mcpeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. on Progr. Lang. and Syst.*, vol. 27, 2005.

[3] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Third International Workshop on Automated Debugging*, 1997.

[4] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of NDSS Symposium*, Feb. 2004.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *IEEE S&P*, 2008.

[6] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for c programs," in *AsiaCCS*, 2010.

[7] C. Cowan, C. Pu, D. Maier, H. Hintongif, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[8] T. cker Chiueh and F. hau Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *ICDCS*, 2001.

[9] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics," in *ACSAC*, 2010.

[10] P. Team, "Address Space Layout Randomization," http://pax.grsecurity.net/docs/aslr.txt, March 2003.

[11] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX Security Symposium*, August 2005.

[12] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovi, "Randomized instruction set emulation," *ACM TISSEC*, 2005.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," Oct. 2003.

[14] T. de Raadt, "Exploit Mitigation Techniques (in OpenBSD, of course)," http://www.openbsd.org/papers/ven05-deraadt/, Nov. 2005.

[15] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003," September 2006.

[16] SANS, "CWE/SANS TOP 25 Most Dangerous Software Errors," http://www.sans.org/top25-software-errors/, Jun 2011.

[17] Symantec, "Symantec report on the underground economy," 2008.

[18] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks," in *DIMVA*, July 2009.

[19] H. S. Ryan Roemer, Erik Buchanan and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM TISSEC*, Apr 2010.

[20] Y. Younan, W. Joosen, and F. Piessens, "Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW386, July 2004.

[21] H. Meer, "Memory Corruption Attacks The (almost) Complete History," in *Blackhat USA*, July 2010.

[22] S. Alexander, "Defeating compiler-level buffer overflow protection," *;login: The USENIX Magazine*, vol. 30, no. 3, July 2005.

[23] J. P. Anderson, "Computer Security Technology Planning Study. Volume 2," Oct. 1972.

[24] C. Schmidt and T. Darby, "The What, Why, and How of the 1988 Internet Worm," July 2001.

[25] CERT Coordination Center, "The CERT FAQ," Jan. 2011.

[26] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *CACM*, vol. 33, no. 12, Dec. 1990.

[27] K. Seifried and E. Levy, "Interview with Elias Levy (Bugtraq)," 2001.

[28] T. Lopatic, "Vulnerability in NCSA HTTPD 1.3," February 1995.

[29] P. Zatko, "How to write Buffer Overflows," 1995.

[30] Aleph1, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 49, no. 14, Nov. 1996.

[31] S. Designer, "Linux kernel patch to remove stack exec permission," http://seclists.org/bugtraq/1997/Apr/31, April 1997.

[32] M. Conover and w00w00 Security Team, "w00w00 on Heap Overflows," http://www.cgsecurity.org/exploit/heaptut.txt, Jan. 1999.

[33] T. Twillman, "Exploit for proftpd 1.2.0pre6," September 1999.

[34] "CVE-2001-1342," May 2001.

[35] M. Dowd, "Application-Specific Attacks: Leveraging the ActionScript Virtual Machine," April 2008.

[36] S. Designer, "Non-executable stack patch," June 1997.

[37] ——, "Linux kernel patch from the Openwall Project."

[38] ——, "Getting around non-executable stack (and fix)," August 1997.

[39] Nergal, "The Advanced Return-Into-Lib(c) exploits (PaX Case study)," *Phrack Magazine*, vol. 58, no. 4, Dec. 2001.

[40] J. McDonald, "Defeating Solaris/SPARC Non-Executable Stack Protection)," March 1999.

[41] The Pax Team, "Design & Implementation of PAGEEXEC," 2000.

[42] T. de Raadt, "The OpenBSD 3.3 Release," May 2003.

[43] A. van de Ven, "New Security Enhancements in Red Hat Enterprise Linux v.3, update 3," August 2004.

[44] I. Molnar, "Exec Shield," May 2003.

[45] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," September 2005.

[46] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *RAID*, Menlo Park, CA, September 2011.

[47] C. Cowan, "StackGuard: Automatic Protection From Stack-smashing Attacks," Dec. 1997.

[48] StackShield, "Stack Shield: A "stack smashing" technique protection tool for Linux," Dec. 1999.

[49] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack Magazine*, vol. 56, no. 5, Jan. 2000.

[50] G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection," June 2002.

[51] D. Litchfield, "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server," in *Blackhat Asia*, Dec. 2003.

[52] M. Miller, "Preventing the Exploitation of SEH Overwrites," September 2006.

[53] Microsoft, "/SAFESEH."

[54] B. Bray, "Compiler Security Checks In Depth," February 2002.

[55] H. Etoh and K. Yoda, "Protecting from stack-smashing attacks," June 2000.

[56] DilDog, "L0pht Advisory MSIE4.0(1)," Jan. 1998.

[57] S. Designer, "JPEG COM Marker Processing Vulnerability," July 2000.

[58] MaXX, "VUDO Malloc Tricks," *Phrack Magazine*, August 2001.

[59] Anonymous, "Once Upon a Free()," *Phrack Magazine*, August 2001.

[60] H. Flake, "Third Generation Exploits," in *Blackhat USA Windows Security*, February 2002.

[61] jp, "Advanced Doug lea's malloc exploits," *Phrack Magazine*, vol. 61, no. 6, August 2003.

[62] D. Litchfield, "Windows Heap Overflows," in *Blackhat USA Windows Security*, Jan. 2004.

[63] M. Conover and O. Horovitz, "Windows Heap Exploitation (Win2KSP0 through WinXPSP2)," in *SyScan*, Dec. 2004.

[64] A. Anisimov, "Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass," Jan. 2005.

[65] N. Falliere, "Critical Section Heap Exploit Technique," August 2005.

[66] B. Moore, "Exploiting Freelist[0] on XP SP2," Dec. 2005.

[67] M. Conover, "Double Free Vulnerabilities," Jan. 2007.

[68] J. McDonald and C. Valasek, "Practical Windows XP/2003 Heap Exploitation," in *Blackhat USA*, July 2009.

[69] P. Phantasmagoria, "Exploiting the wilderness," February 2004.
[70] A. Marinescu, "Windows Vista Heap Management Enhancements," in *Blackhat USA*, August 2006.
[71] N. Waisman, "Understanding and Bypassing Windows Heap Protection," June 2007.
[72] B. Hawkes, "Attacking the Vista Heap," in *Blackhat USA*, August 2008.
[73] P. Phantasmagoria, "The Malloc Maleficarum," Oct. 2005.
[74] blackngel, "Malloc Des-Maleficarum," *Phrack Magazine*, June 2009.
[75] ——, "The House Of Lore: Reloaded," *Phrack Magazine*, vol. 67, no. 8, Nov. 2010.
[76] BugTraq, "Wu-Ftpd Remote Format String Stack Overwrite Vulnerability," June 2000.
[77] Scut, "Exploiting Format String Vulnerabilities," September 2001.
[78] G. . Riq, "Advances in format string exploitation," *Phrack*, July 2002.
[79] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, "FormatGuard: Automatic Protection From printf Format String Vulnerabilities," in *USENIX Security Symposium*, August 2001.
[80] Microsoft, "Disable `%n` format string," 2010.
[81] C. Planet, "A Eulogy for Format Strings," *Phrack*, Nov. 2010.
[82] T. de Raadt, "The OpenBSD Project," http://cansecwest.com/core03/theo-csw03.mgp, April 2003.
[83] ——, "The OpenBSD 3.4 Release," Nov. 2003.
[84] greydns and T. de Raadt, "OT: PaX question," April 2003.
[85] P. Team, T. de Raadt, and D. Schellekens, "[OT] PaX," April 2003.
[86] D. Schellekens, P. Team, and Anonymous, "Recent OpenBSD changes vs PaX," April 2003.
[87] B. Spengler, "PaX: The Guaranteed End of Arbitrary Code Execution," Oct. 2003.
[88] A. van de Ven, "Patch 0/6 virtual address space randomisation," Jan. 2005.
[89] ——, "Changes from v2.6.11 to v2.6.12-rc1," March 2005.
[90] L. Torvalds, "Linux 2.6.25 ChangeLog," April 2008.
[91] D. Calleja, "Linux 2.6.25," May 2008.
[92] M. Howard, "Address Space Layout Randomization in Windows Vista," May 2006.
[93] A. Rahbar, "An analysis of Microsoft Windows Vista's ASLR," Oct. 2006.
[94] M. Howard, "Alleged Bugs in Windows Vistas ASLR Implementation," Oct. 2006.
[95] O. Whitehouse, "An Analysis of Address Space Layout Randomization on Windows Vista," March 2007.
[96] A. R. Pop, "DEP/ASLR Implementation Progress in Popular Third-party Windows Applications," June 2010.
[97] A. Inc., "Mac OS X Leopard Security," Oct. 2007.
[98] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *USENIX Security Symposium*, August 2003.
[99] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A Defense Against Heap-spraying Code Injection Attacks," in *Proceedings of the USENIX Security Symposium*, August 2009.
[100] T. Durden, "Bypassing PaX ASLR Protection," *Phrack Magazine*, vol. 59, no. 9, July 2002.
[101] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *ACM CCS*, Oct. 2004.
[102] T. Müller, "ASLR Smack & Laugh Reference," February 2008.
[103] FHM Crew, "ASLR bypassing method on 2.6.17/20 Linux Kernel," September 2008.
[104] G. Fresi-Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *ACSAC*, Dec. 2009, pp. 60–69.
[105] SkyLined, "Internet Explorer IFRAME src&name parameter BoF remote compromise," Oct. 2004.
[106] ——, "Internet Exploiter 3: Technical details," Nov. 2004.
[107] ——, "Microsoft Internet Explorer DHTML Object handling vulnerabilities (MS05-20)," April 2005.
[108] A. Sotirov, "Heap Feng Shui in JavaScript," in *Blackhat Europe*, March 2007.
[109] D. Blazakis, "Interpreter Exploitation: Pointer Inference and JIT Spraying," in *Blackhat DC*, July 2010.
[110] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *ACM CCS*, 2010.
[111] J. Salwan, "ROPgadget tool v3.3," Nov. 2011.
[112] E. Buchanan, R. Roemer, H. Shacham, , and S. Savage, "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," in *ACM CCS*, Oct 2008.
[113] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *USENIX Security Symposium*, 2006.
[114] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified Process Replicae for Defeating Memory Error Exploits," in *Intern. Workshop on Assurance (WIA)*, 2007.
[115] B. Salamat, T. Jackson, A. Gal, and M. Franz., "Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space," in *EuroSys*, 2009.
[116] B. Jack, "Vector Rewrite Attack: Exploitable NULL Pointer Vulnerabilities on ARM and XScale Architectures," 2007.
[117] M. M. (skape), "Exploiting the Otherwise Non-exploitable on Windows," 2006.
[118] "CVE-2007-0071," April 2008.
[119] J. Tinnes and T. Ormandy, "Bypassing Linux' NULL pointer dereference exploit prevention (mmap_min_addr)," June 2009.
[120] B. Spengler, "Enlightenment," September 2009.
[121] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. on Program. Lang. and Syst.*, July 2010.
[122] J. Corbet, "Fun with NULL pointers," July 2009.
[123] D. Walsh, "Confining the unconfined," July 2009.
[124] J. Tinnes and T. Ormandy, "Linux NULL pointer dereference due to incorrect proto_ops initializations," August 2009.
[125] M. J. Cox, "Red Hat's Top 11 Most Serious Flaw Types for 2009," http://www.awe.com/mark/blog/20100216.html, Feb 2010.
[126] A. Baratloo, T. Tsai, and N. Singh, "Libsafe: Protecting Critical Elements of Stacks," Dec. 1999.
[127] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Sec. Symposium*, 2005.
[128] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities," August 2003.
[129] Blexim, "Basic Integer Overflows," *Phrack*, Dec. 2002.
[130] M. Dowd, C. Spencer, N. Metha, N. Herath, and H. Flake, "Advanced Software Vulnerability Assessment," in *Blackhat USA*, August 2002.
[131] O. Horovitz, "Big Loop Integer Protection," *Phrack*, Dec. 2002.
[132] D. Brumley, T. cker Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically Protecting Against Integer-Based Vulnerabilities," in *In Symp. on Network and Distributed Systems Security*, March 2007.
[133] T. Wang and Z. Lin, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution," in *In Symp. on Network and Distributed Systems Security*, February 2009.
[134] M. Howard, "Safe Integer Arithmetic in C," February 2006.
[135] D. LeBlanc, "Integer Handling with the C++ SafeInt Class," urlhttp://msdn.microsoft.com/en-us/library/ms972705, Jan. 2004.
[136] ——, "SafeInt 3 on CodePlex!" September 2008.
[137] ——, "Safeint," http://safeint.codeplex.com/, March 2011.
[138] F. von Leitner, "Catching Integer Overflows in C," Jan. 2007.
[139] S. Christey and R. A. Martin, "Vulnerability Type Distributions in CVE," May 2007.
[140] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *CACM*, vol. 27, no. 1, 1984.
[141] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ISSTA*, 2002.
[142] K. Zetter, "Three minutes with rain forrest puppy," 2001.
[143] D. Goodin, "Legal goons threaten researcher for reporting security bug," 2011.
[144] R. Lemos, "Does Microsoft Need Bug Bounties?" May 2011.
[145] D. Fisher, "Survey Shows Most Flaws Sold For $5,000 Or Less," May 2010.
[146] VUPEN, "Google Chrome PWNED on Windows, exploit leaps over sandbox/ASLR/DEP," May 2011.
[147] S. Fewer, "Pwn2Own 2011: IE8 on Windows 7 hijacked with 3 vulnerabilities," May 2011.
[148] VUPEN, "Safari/MacBook first to fall at Pwn2Own 2011," March 2011.
[149] NIST, "National Vulnerability Database."
[150] Symantec, "Vulnerability Trends."
[151] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf, "Mobile security catching up? - revealing the nuts and bolts of the security of mobile devices," in *IEEE S&P*, 2011.

[152] M. BlueHat, “Microsoft BlueHat Prize Contest,” 2011.