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

Victor van der Veen[1], Nitish dutt-Sharma[1], Lorenzo Cavallaro[1,2], and Herbert Bos[1]

[1] The Network Institute, VU University Amsterdam
[2] 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 insights in past trends and developments, while an investigation of real-world vulnerabilities and exploits allows us to answer on the significance of memory errors in the foreseeable future.

1 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. From safe languages, which remove the vulnerability entirely [53,72], and bounds checkers, which check for out-of-bounds accesses [3,54,82,111], to countermeasures that prevent certain memory locations to be overwritten [25,29], detect code injections at early stages [80], or prevent attackers from finding [11,98], using [8,56], or executing [32,70] 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 [85]. Experience shows that attackers, motivated nowadays by profit rather than fun [97], have been effective at finding ways to circumvent protective measures [39,83]. 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

* This work was partially sponsored by the EU FP7 SysSec project and by an ERC Starting Grant project ("Rosetta").

solved as "dozens of commercial solutions exist" and research should focus on other problems, while another questioned the usefulness of the research efforts, as they clearly "could 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 [110], we provide the reader with a *comprehensive bird-eye view* and *analysis* on the matter. This paper strives to be *the* reference on memory errors.

To this end, we first present (Section 2) 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. Like Miller et al. [68], we use a compact timeline to drive our discussion, but categorize events in a more structured way, based on a branched timeline.

Second, we present a study of memory errors statistics, analyzing vulnerabilities and exploit occurrences over the past 15 years (Section 3). 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 trend. A tentative conclusion is that memory errors are unlikely to lose much significance in the near future and that perhaps it is time adopt a different mindset, where a number of related research areas are explored, as suggested in Section 4. We conclude in Section 5.

2 A High Level View of Memory Error History

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 [5]. However, it was only after more than a decade that this concept was further developed. On November the 2nd, 1988, the Internet Worm developed by Robert T. Morris abruptly brought down the Internet [86]. The worm exploited a buffer overflow vulnerability in `fingerd`.

In reaction to this catastrophic breach, the Computer Emergency Response Team Coordination Center (CERT/CC) was then formed [22]. CERT/CC's main goal was to collect user reports about vulnerabilities and forward them to vendors, who would then take the appropriate action.

In response to the lack of useful information about security vulnerabilities, Scott Chasin started the Bugtraq mailing list in November 1993. At that time,

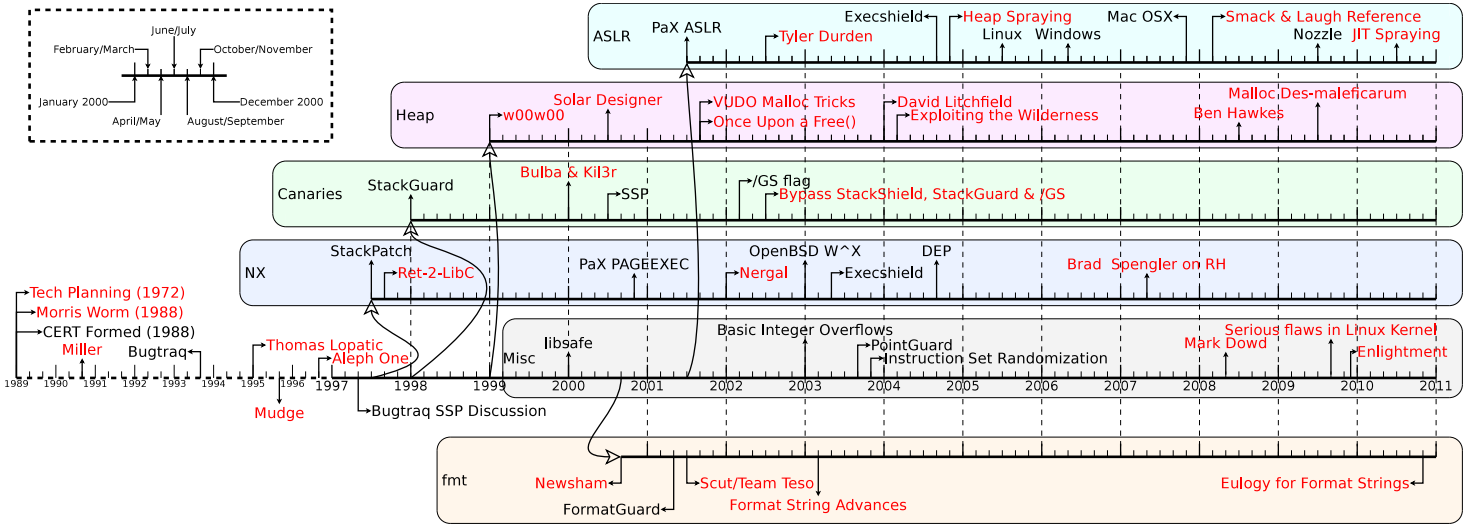


Fig. 1: General timeline

many considered the CERT/CC of limited use, as it could take years before vendors released essential patches. In contrast, Bugtraq offered an effective tool to *publicly* discuss on the subject, without relying on vendors' responsiveness [88].

In 1995, Thomas Lopatic boosted the interest in memory errors by describing a step-by-step exploitation of an error in the NCSA HTTP daemon [63]. Shortly after, Peiter Zatko (Mudge) released a private note on how to exploit the now classic memory error: stack-based buffer overflows [112]. So far, nobody really discussed memory error countermeasures, but after Mudge's notes and the well-known document by Elias Levy (Aleph One) on stack smashing [4], discussions on memory errors 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 [34], in April 1997. We discuss NX in Section 2.1.

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 [29]. Further details are discussed in Section 2.2.

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 [27], which we discuss in Section 2.3.

On September 20, 1999, Tymm Twillman introduced format string attacks by posting an exploit against ProFTPD on Bugtraq [101]. Format string exploits became popular in the next few years and we discuss them in Section 2.4.

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. We discuss the various types of ASLR and its related attacks in Section 2.5.

Around the same time as the introduction of ASLR, another type of vulnerability, `NULL` pointer dereference, a form of dangling pointer, was disclosed in May 2001. Many assumed that such dangling pointers were unlikely to cause more harm than a simple denial of service attacks. In 2007 and 2008, however, Afek and Sharabani and Mark Dowd showed that these vulnerabilities could very well be used for arbitrary code injection as well [1,37]. Unfortunately, specific defenses against dangling pointers are still mostly research-driven efforts [2].

Due to space limitations, a number of historical details were omitted in this paper. The interested reader can refer to [102] for more information.

2.1 Non-Executable Stack

Stack-based buffer overflows are probably the best-known memory error vulnerabilities [4]. 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 return address is reached. The corruption of this code pointer (making it point to the buffer itself, partially filled with

attacker-injected code) allows the execution of 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 [34].

Just a few months after introducing the patch, Solar Designer himself described a novel attack that allows attackers to bypass a non-executable stack [33]. 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 library can be the target of this diversion, the attack is dubbed *return-into-libc* because the return address is typically replaced with the address of proper C library functions (and arguments).

An enhancement of Solar Designer's non-executable stack was quickly proposed to withstand return-into-libc attacks [33]. However, shortly thereafter, Rafal Wojtczuk (Nergal) circumvented 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) [73]. McDonald [66] built on such results and proposed return-into-libc as a first stage loader to run the injected code in a non-executable segment.

The PaX Team went far beyond a non-executable stack solution. With the PaX project released in the year 2000 [99], 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 CPU lacks this feature, PaX can emulate it in software. In addition, PaX randomizes the mmap base so that both the process' stack and the first loaded library will be mapped at a random location, representing the first form of address space layout randomization (Section 2.5).

One of the first attacks against PaX ASLR was published by Nergal [73] 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 stack from the Linux proc file system for a local exploit. Moreover, if the attacker can provide the payload from I/O pipes rather than the environment or arguments, then the program is exploitable.

OpenBSD version 3.3, released in May 2003, featured various memory error mitigation techniques. Among these, W^X proved to be effective against code-injection attacks. As a memory page can either be writable or executable, but never be both, injected code had no chances to get executed anymore.

In August 2005, Microsoft released the Service Pack 2 (SP2) of the Windows XP OS, featuring Data Execution Protection (DEP)—which prevents code execution of a program data memory [70].

By this time all major OSes were picking up on memory error mitigation techniques. NX 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 NX can only partially mitigate memory errors from being exploited.

In 2005, Krahmer [58] pioneered short code snippet reuse instead of entire libc functions for exploit functionality—a direction that reached its zenith in return-oriented programming (ROP). Attackers chain code snippets together to create *gadgets* that perform predetermined but arbitrary computations [83]. The chaining works by placing short sequences of data on the stack that drive the flow of the program whenever a call/return-like instruction executes.

To date, no ROP-specific countermeasures have seen deployment in mainstream OSes. Conversely, low-overhead bounds checkers [3,111] and practical taint-tracking [16] may be viable solutions to defeat control-hijacking attacks.

2.2 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 of such systems, StackGuard, was released on January 29, 1999 [29]. 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 [96], released later in 1999, tried to address this issue by focusing on the return address itself, by copying it to a "secure" location. Upon function termination, the copy is checked against the actual return address. A mismatch would result in program termination.

StackShield clearly showed 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 overflows (e.g., dynamic memory allocator metadata corruption) and format bug vulnerabilities intermix (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 [21,81]. On Windows, David Litchfield introduced a novel approach to bypass canary-based protections by corrupting specific exception handling callback pointers, i.e., structured exception handing (SEH), used during program cleanup, when a return address corruption is detected [61].

Matt Miller subsequently proposed a protection against SEH exploitation [71] 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 the terminator unreachable, they are often easy to detect. Unlike alternative solutions introduced by Microsoft [17], 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.

Despite their initial weaknesses, canary-based protection spun off more countermeasures. ProPolice, known also as Stack Smashing Protection (SSP), built on the initial concept of StackGuard but addressed its shortcomings [40]; stack variables are rearranged such that pointers corruptions due to buffer overflows are no longer possible. SSP was successfully implemented as a low-overhead patch for the GNU C compiler and was included in mainstream from version 4.1.

2.3 Protecting the Heap

While defense mechanisms against stack-based buffer overflow exploitations were deployed, heap-based memory errors were not taken into consideration yet.

The first heap-based buffer overflow can be traced to January 1998 [36], and the first paper published by the underground research community on heap-based vulnerabilities appeared a year later [27]. While 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 exploits was reported by Solar Designer in July 2000 [35]. The exploit shows that in-band control information (heap management metadata) is still the issue, a bad practice, and should *always* be avoided, unless robust integrity checking mechanisms are in place. Detailed public disclosures of heap-based exploitations appeared in [7,65]. 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 reliable heap-based memory error exploits.

Windows OSes were not immune from heap exploitation either. BlackHat 2002 hosted a presentation by Halvar Flake on the subject [45], while more advanced UNIX-based heap exploitation techniques where published in August 2003 [55], describing how to obtain a *write-anything-anywhere* primitive that, along with information leaks, allow for successful exploits even when ASLR is in use.

More about Windows-based heap exploitations followed in 2004 [62]. The introduction of Windows XP SP2, later that year, came with a non-executable heap. In addition, SP2 introduced heap cookies and safe heap management metadata unlink operations. Not long had to be waited for before seeing the first working exploits against Microsoft latest updates [26,6,67]. With the release of Windows Vista in January 2007, Microsoft further hardened the heap against exploitation [64]. However, as with the UNIX counterpart, there were situations in which application-specific attacks against the heap could still be executed [51,107].

In 2009 and 2010 a report appeared where proof of concept implementations of almost every scenario described in [78] were shown in detail [12,13].

2.4 Avoiding Format String Vulnerabilities

Similarly to the second generation of heap attacks, but unlike classic buffer overflows, format string vulnerabilities (also known as format bugs) 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* reads of the whole process address space. Disclosing confidential data (e.g., cryptographic material and secrets [29,98]), executing arbitrary code, and exploring the whole address space of a victim process are all viable possibilities.

Format string vulnerabilities were first discovered in 1999 while auditing ProFTPD [101], but it was in the next couple of years that they gained popularity. A format string vulnerability against WU-FTPD was disclosed on Bugtraq in June 2000 [20], while Tim Newsham was the first to dissect the intricacies of the attack, describing the fundamental concepts along with various implications of having such a vulnerability in your code.

One of the most extensive articles on format string vulnerabilities was published by Scut of the TESO Team [87]. Along with detailing conventional format string exploits, he also presented novel hacks to exploit this vulnerability.

Protection against format string attacks was proposed by FormatGuard in [28]. It 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`-like directives by default. Similarly, the GNU C library `FORTIFY_SOURCE` patches provide protection mechanisms, which make format string exploitations hard. Even so, and although the low hanging fruit had been harvested long ago, the challenge of breaking protection schemes remains exciting [79].

2.5 Address Space Layout Randomization

Memory error exploitations typically require an intimate knowledge of the address space layout of a process to succeed. Therefore, any attempt to randomize that layout would increase the resiliency against such attacks.

The PaX Team proposed the first form of address space layout randomization (ASLR) in 2001 [99]. ASLR can be summarized succinctly as the introduction of randomness in the address space layout of userspace processes. Such randomness would make a class of exploits fail with a quantifiable probability.

PaX-designed ASLR underwent many improvements over time. The first ASLR implementation provided support for `mmap` base randomization (July 2001). When randomized `mmap` base is enabled, dynamically-linked code is mapped starting at a different, randomly selected base address each time a program starts, making return-into-libc attacks difficult. Stack-based randomization followed quickly in August 2001. Position-independent executable (PIE) randomization was proposed in the same month. A PIE binary is similar in spirit to

dynamic shared objects as it can be loaded at arbitrary addresses. This reduces the risk of performing successful return-into-plt [73] or more generic return-oriented programming attacks [83] (see next). The PaX Team proposed a kernel stack randomization in October 2002 and, to finish their work, a final patch was released to randomize the heap of processes.

Over time, OSes deployed mostly coarse-grained—often kernel-enforced—forms of ASLR, without enabling PIE binaries. 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 starting base addresses are randomized, while relative offsets (e.g., the location of any two objects in the process address space) are fixed. Thus, an attacker's task is to retrieve the absolute address of a *generic* object of, say, a dynamically-linked library of interest: any other object (e.g., library functions used in return-into-libc attacks) can be reached as an offset from it.

One of the first attacks against ASLR was presented by Nergal in 2001 [73]. 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 a direct call to 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 [38]. 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, the available entropy is generally limited and leaves brute forcing attacks a viable alternative to exploit ASLR-protected systems [90].

Finally, Fresi-Roglia et al. [47] detail a return-oriented programming [83] attack able to bypass W^X and ASLR. This attack chains code snippets of the original executable 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-libc 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.

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 [91,92,93]. By populating the heap with a large number of objects containing attacker-provided code, he made it possible to increase the likelihood of success in referencing (and executing) it.

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.

Dion Blazakis went far beyond heap spraying by describing pointer inference and JIT spraying techniques [14]. Wei et al. proposed dynamic code generation (DCG) spraying, a generalized and improved JIT spraying technique [108]. (Un)luckily DCG suffers from the fact that memory pages, which are about to contain dynamically-generated code, have to be marked as being writable *and* executable. 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 was eventually proposed [108].

Finally, return-oriented programming, introduced in Section 2.1, may also be used to bypass non-PIE ASLR-protected binaries (as shown by [47]). In fact, for large binaries, the likelihood of finding enough useful code snippets to build a practical attack is fairly high. Recent work on protecting against these attacks involves instruction location randomization, in-place code randomization and fine-grained address space randomization [48,52,77].

3 Data Analysis

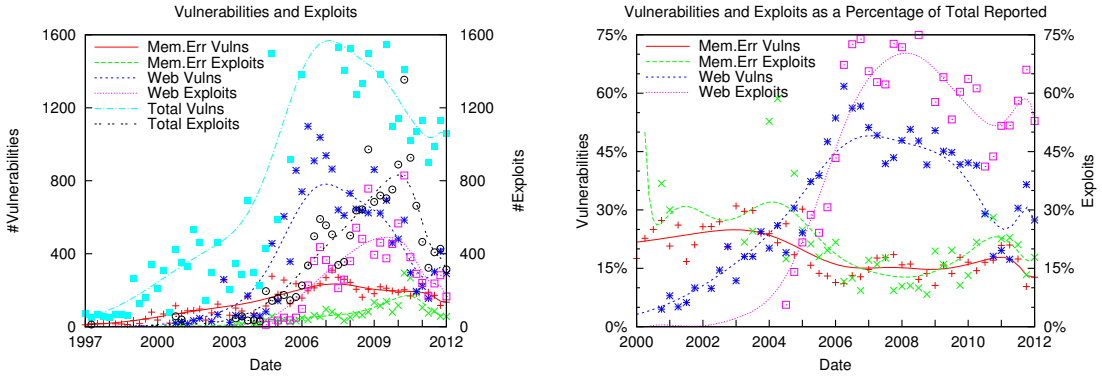
We have analyzed statistics as well as real-life evidence about vulnerability and exploit reports to draw a final answer about memory errors. To this end, we tracked vulnerabilities and exploits over the past 15 years by examining the Common Vulnerabilities and Exposures (CVE) and ExploitDB databases.

Figure 2a 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. Conversely, the downward trend in discovered vulnerabilities that started in 2007 is remarkable. Instead of a linear growth, it seems that the number of *reported* vulnerabilities is now reversed. It is worth noting that such a drop mirrors a similar trend in the *total* number of reported vulnerabilities. Figure 2b shows the same results as percentages.

The spike in the number of vulnerabilities that started in 2003 may well have been caused by the explosive growth of web vulnerabilities in that period (as supported by [24]).

Figure 2a shows that web vulnerabilities first appeared in 2003 and rapidly outgrew the number of buffer overflow vulnerabilities. Probably due to their simplicity, the number of working web exploits also exceeded the number of buffer overflow exploits in 2005. It is therefore plausible that the extreme growth in vulnerability reports that started in 2003 had a strong and remarkable web-related component. This seems to be reasonable as well: shortly after the dot-com bubble in 2001, when the Web 2.0 started to kick in, novel web developing technique were often not adequately tested against exploitation techniques. This was probably due to the high rate at which new features were constantly asked by end customers: applications had to be deployed quickly to keep up with competitors. This race left little time to carefully perform code security audits.

As mentioned earlier, Figure 2a also shows a downward trend in the total number of vulnerabilities over the years 2006–2010, as reported independently



(a) Absolute (b) As % of totals

Fig. 2: Vulnerabilities and Exploits.

in [68]. Memory errors were also affected by that trend and started to diminish in early 2007. Despite such negative fluctuations, generic and memory error-specific *exploits* kept growing linearly each month. The reasons for the downward trend of reported vulnerabilities could be manifold; it could be that fewer bugs were found in source code, fewer bugs were reported, or 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 up the first hypothesis. More LoC results in more bugs: software reliability studies have shown that executable code may contain up to 75 bugs per 1000 LoC [9,76]. CVEs look at vulnerabilities and do not generally make a difference between plain vulnerabilities and vulnerabilities that could be exploited. Memory error mitigation techniques are unlikely to have contributed to the drop in reported vulnerabilities. Most defense mechanisms that we have discussed earlier do not result in safer LoC (for which static analysis can instead help [103]); they only prevent exploitation of vulnerable—usually poorly written—code.

However, if we look carefully at the data provided by analyzing CVE entries, as depicted in Figure 2a, we see that the number of web vulnerabilities follows the same trend as that of the total number of vulnerabilities. It seems that both the exponential growth (2003–2007) and drop (2007–2010) in vulnerabilities is related to fundamental changes in web development. It is reasonable to assume that companies, and especially web developers, started to take web programming more seriously in 2007. For instance, developers may well have become more aware of attacks such as SQL injections or Cross Site Scripting (XSS). If so, this would have raised web security concerns, resulting in better code written.

Similarly, static (code) analysis may have started to be included in the software development life-cycle. Static code analysis for security tries to find weaknesses in a program that might lead to security vulnerabilities. In general, the

process is to evaluate a system (and all its components) based on its form, structure, content or documentation for non-conformance in access control, information flow, or application programming interface. Considering the high cost involved in manual code review, software industry prefers using automated code analyzers. IBM successfully demonstrated JavaScript analysis [109] by analysing 678 websites (including 178 most popular websites). Surprisingly, 40% of the websites were found vulnerable and 90% of vulnerable applications had 3rd party code. Another interesting take on static analysis can be observed in the surveys published by NIST [75,74]. A remarkable number of previously reported vulnerabilities were found in popular open-source programs using a combination of results reported by different tools. For instance, [103] reports intriguing figures showing the industry has confidence in static analysis. The software and IT industry are the biggest requester, followed by the finance sector for independent security assessments of their applications. Furthermore, 50% of the companies resubmitted 91–100% of their commercial application (after the first submissions revealed security holes) for code analysis. The growing trust of industry in static analysis could be one of the reasons for a drop in the number of reported vulnerabilities from the last few years.

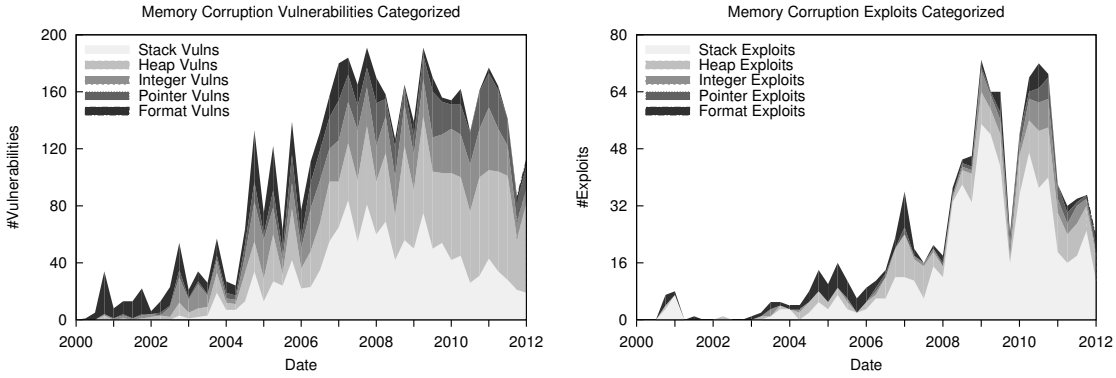
To substantiate the second hypothesis (i.e., less bugs are reported), it is necessary and helpful to have a more social view on the matter. There could be a number of reasons why people stopped reporting bugs to the community.

Empirical evidence about "no full disclosure due to bounties" advocates this statement very well. Ten years ago, the discovery of a zero-day vulnerability would have likely had a patch and a correspondence with the application authors/vendor about the fix, likely on public mailing lists. Today, big companies like Google and Mozilla give out rewards to bug hunters as long as they do not disclose the vulnerability [69]. Similarly, bug hunters may choose to not disclose zero day vulnerabilities in public, but sell them instead to their customers [43].

Where companies send out rewards to finders of vulnerabilities, useful zero-days could yield even more in underground and private markets. This business model suggests that financial profit may have potentially been responsible for the downward trend. While more and more people start buying things online and use online banking systems, it becomes increasingly interesting for criminals to move their activities to the Internet as well. Chances that issues found by criminals are reported as CVEs are negligible.

At the same time, full disclosure [113] as it was meant to be, is being avoided [50,44]. As an example for this shift in behavior, researchers were threatened for finding a vulnerability [49], or, as mentioned above, they may well sell them to third parties on private markets. This was also recently backed up by Lemos and a 2010-survey that looked at the relative trustworthiness and responsiveness of various organizations that buy vulnerabilities [60,42].

In conclusion, it is reasonable to believe that the drop in vulnerabilities is caused by both previous hypotheses. 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



(a) Vulnerabilities (b) Exploits

Fig. 3: Memory errors categorized.

to more professional platforms instead of their home-brew frameworks and eliminated easy vulnerabilities by simply writing better code. This growing professionalism of the software industry also contributed to the fact that bugs are no longer reported to the public, but sold to either the program's owners or the underground economy.

3.1 Categorizing Vulnerabilities and Exploits

We further categorized memory error vulnerabilities and exploits in 6 different classes (based on their CVEs descriptions): stack-based, heap-based, integer issues, `NULL` pointer dereference and format string. Figures 3a and 3b show the classification for vulnerabilities and exploits, respectively.

From Figures 3a and 3b we make the following observations which may help to draw our final conclusions. 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 as of this writing (see [102]). Last, old-fashioned *stack* and *heap* memory errors are still by far (about 90%) *the most exploited* ones, counting for about 50% of all the reported vulnerabilities. There is no evidence to make us believe this will change in the near future.

4 Discussion

To answer the question whether memory errors have become a memory of the past, a few more observations need to be taken into consideration.

Table 1: Breakdown of exploited vulnerabilities in popular exploit toolkits.

Pack	Exploits	Memory Errors	Unspecified	Other	Updated
Nuclear	12	6 (50%)	3 (25%)	0 (0%)	Mar. 2012
Incognito	9	4 (44%)	1 (33%)	0 (0%)	Mar. 2012
Phoenix	26	14 (54%)	7 (27%)	5 (19%)	Mar. 2012
BlackHole	15	6 (40%)	7 (46%)	2 (13%)	Dec. 2011
Eleonore	31	18 (58%)	6 (19%)	7 (23%)	May. 2011
Fragus	14	11 (79%)	1 (7%)	2 (14%)	2011
Breeding Life	15	8 (53%)	6 (40%)	1 (6%)	?
Crimepack	19	10 (53%)	2 (11%)	7 (37%)	Jul. 2010
All	104	66 (63%)	12 (12%)	26 (25%)	

Impact. Let us first take a closer look at the impact of memory error vulnerabilities. After all, if this turns out to be negligible, then further research on the topic may just well be a questionable academic exercise. To provide a plausible answer, we analysed different exploit packs by studying the data collected and provided by contagio malware dump[3]. Table 1 shows the number of exploits (with percentages too) a given exploit pack supports and it is shipped with. The column Memory Errors reports those related to memory errors, while Unspecified refers to exploit of vulnerabilities that have not been fully disclosed yet (and chances are that some of these are memory error-related, e.g., CVE-2010-0842). Conversely, the column Other refers to exploits that are anything but memory error-related (e.g., an XSS attack).

Table 1 clearly shows that in at least 63% of the cases, memory error exploits have been widely deployed in exploit packs and have thus a large impact on the security industry, knowing that these exploit packs are responsible for large-scale hosts infections.

Support of buffer overflows by design. Second, we observe that the number of memory vulnerabilities in a specific program is highly dependent on the programming language of choice. Looking closely at the C programming language, we observe that it actually needs to support buffer overflows. Consider, for example, an array of a simple C struct containing two fields, as depicted in Figure 4a. A `memset()`-like call may indeed overflow a record (Figure 4b). In this case, the overflow is not a programming error, but a desired action. Having such overflows by design, makes the programming language more vulnerable and harder to protect against *malicious* overflows. Considering that unsafe programming languages such as C and C++ are and have been among the most popular languages in the world for a long time already (as supported by the TIOBE Pro-

[3] http://contagiodump.blogspot.com/2010/06/overview-of-exploit-packs-update.html

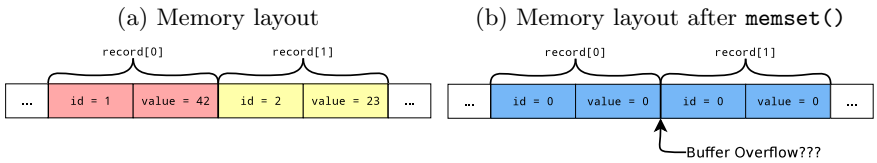


Fig. 4: Buffer overflow support in C.

gramming Community Indexes), careful attention should be paid by developers to avoid memory error vulnerabilities.

Deployment of mitigation techniques. Third, we observe that mitigation techniques are not always deployed by modern OSes. The reasons could be manifold. For instance, implementing a specific mitigation technique on legacy (or embedded) systems may require non-existent hardware support, or it may incur non-negligible overheads. Similarly, alternative mitigation techniques may require recompilation of essential parts of a system, which cannot be done due to uptime requirements or lack of source code.

Patching behaviour. Another issue relates to the patching behaviour. Not only end users, but also system administrators appear to be lazy when it comes to patching vulnerabilities or updating to newer software versions. During our research on exploit kits, we found that even recently updated exploit packs still come with exploits for quite dated vulnerabilities, going back to 2006. This is backed up by other studies [59,100].

Motivation. Finally, even when all mitigation techniques are deployed, skilled and well-motivated attackers can still find their way into a system [41,104,106,105].

Looking back at Figure 2a, it is reasonable to say that some form of awareness has arisen among developers. On the other hand, Figure 2b shows that memory errors have a market share of almost 20%—a number that did not change much over the last 15 years, and something of which we have no evidence that it is about to change in the foreseeable future. The fact that over 60% of all the exploits reported in Table 1 are memory error-related, does not improve the scenario either.

4.1 Research Directions

Memory errors clearly still represent a threat undermining the security of our systems. We would like to conclude by sketching a few research directions that we consider both important and promising.

Information leakage, function pointer overwrites and heap inconsistencies. Information leakage vulnerabilities are often used to bypass (otherwise well-functioning) ASLR, enabling an attacker to initiate a return-into-libc or ROP attack [89]. Function pointer overwrites and heap inconsistencies form a class of vulnerabilities that cannot be detected by current stack smashing detectors and heap protectors. These vulnerabilities are often exploited to allow

arbitrary code execution [105]. Recent studies try to address these problems by introducing more randomness at the operating system level[77,48,52] and, together with future research, will hopefully result in better protections against these classes of vulnerabilities [14].

Low-overhead bounds checkers. Bounds checking aims at defining the boundaries of memory objects therefore avoiding overflows, which represent probably the most important class of memory errors. Although state-of-the-art techniques have drastically lowered the overhead imposed [111,3], the runtime and memory pressure of such countermeasures are still non-negligible.

Non-control data attacks. While ordinary attacks against control data are relatively easy to detect, attacks against non-control data can be very hard to spot. These attacks were first described by Chen et al. in [23], but a real-world scenario (an attack on the Exim mailserver), was recently described by Sergy Kononenko [57]. Although the attack uses a typical heap overflow, it does not get detected by NX/DEP, ASLR, W^X, canaries nor system call analysis, as it does not divert the program's control flow.

Legacy systems and patching behaviour. As discussed earlier, lazy patching behaviour and unprotected legacy systems (as well as financial gain) are probably the main reasons of the popularity of exploit kits. Sophisticated patching schemes, and disincentivizing automatic patch-based exploit generation [18], should be the focus of further research.

Static and dynamic analysis to detect vulnerabilities. By using novel analysis techniques on vulnerable code, one may succeed in detecting vulnerabilities, and possibly harden programs before the application is deployed in a production environment. Research on this topic is ongoing [94,95] and may help to protect buffer overflow vulnerabilities from being exploited. However, it is necessary to extend these approaches to provide comprehensive protection against memory errors on production environments [46,30,19,84].

Sandboxing. To reduce the potential damage that an exploitable vulnerability could cause to a system, more research is needed on containment mechanisms. This technique is already widely adopted on different platforms, but even in its sophisticated forms (e.g., the Android's sandbox), it comes with weaknesses and ways to bypass it [31].

5 Conclusion

Despite half a century worth 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, and our study, but it is also backed up by evidence showing that even state-of-the-art *detection* and *containment* techniques fail to protect against motivated attackers [41,104]. Besides, protecting mobile applications from memory errors may even be more challenging [10].

Finding alternative mitigation techniques is no longer (just) an academic exercise, but a concrete need of the industry and society at large. For instance, vendors have recently announced consistent cash prizes to researchers who will im-

prove on the state-of-the-art detection and mitigation techniques against memory error attacks [15], showing their concrete commitment towards a long-standing battle against memory error vulnerabilities.

References

1. Afek, J., Sharabani, A.: Dangling Pointer, Smashing the Pointer for Fun and Profit. In: Blackhat USA. (2007)
2. Akritidis, P.: Cling: A memory allocator to mitigate dangling pointers. In: Proceedings of the 19th USENIX conference on Security. (2010)
3. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th conference on USENIX security symposium. (2009)
4. Aleph1: Smashing The Stack For Fun And Profit. Phrack Magazine (Nov. 1996)
5. Anderson, J.P.: Computer Security Technology Planning Study. Volume 2 (Oct. 1972)
6. Anisimov, A.: Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass (Jan. 2005)
7. Anonymous: Once Upon a Free(). Phrack Magazine (Aug. 2001)
8. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanovi, D.: Randomized instruction set emulation. ACM TISSEC (2005)
9. Basili, V.R., Perricone, B.T.: Software errors and complexity: an empirical investigation. CACM (1984)
10. Becher, M., Freiling, F.C., Hoffmann, J., Holz, T., Uellenbeck, S., Wolf, C.: Mobile security catching up? In: IEEE S&P. (2011)
11. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: USENIX Security Symposium. (Aug. 2005)
12. blackngel: Malloc Des-Maleficarum. Phrack Magazine (Jun. 2009)
13. blackngel: The House Of Lore: Reloaded. Phrack Magazine (Nov. 2010)
14. Blazakis, D.: Interpreter Exploitation. In: Proceedings of the 4th USENIX conference on Offensive technologies. (2010)
15. BlueHat, M.: Microsoft BlueHat Prize Contest (2011)
16. Bosman, E., Slowinska, A., Bos, H.: Minemu: The world's fastest taint tracker. In: RAID. (Sept. 2011)
17. Bray, B.: Compiler Security Checks In Depth (Feb. 2002)
18. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy. (2008)
19. Bruschi, D., Cavallaro, L., Lanzi, A.: Diversified Process Replicae for Defeating Memory Error Exploits. In: Intern. Workshop on Assurance (WIA). (2007)
20. BugTraq: Wu-Ftpd Remote Format String Stack Overwrite Vulnerability (Jun. 2000)
21. Bulba, Kil3r: Bypassing StackGuard and StackShield. Phrack Magazine (Jan. 2000)
22. CERT Coordination Center: The CERT FAQ (Jan. 2011)
23. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Sec. Symposium. (2005)
24. Christey, S., Martin, R.A.: Vulnerability Type Distributions in CVE (May 2007)

25. cker Chiueh, T., hau Hsu, F.: Rad: A compile-time solution to buffer overflow attacks. In: ICDCS. (2001)
26. Conover, M., Horovitz, O.: Windows Heap Exploitation (Win2KSP0 through WinXPSP2). In: SyScan. (Dec. 2004)
27. Conover, M., w00w00 Security Team: w00w00 on Heap Overflows (Jan. 1999)
28. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In: USENIX Security Symposium. (Aug. 2001)
29. Cowan, C., Pu, C., Maier, D., Hintongif, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7th USENIX Security Symposium. (Jan. 1998)
30. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: a secretless framework for security through diversity. In: USENIX Security Symposium. (2006)
31. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege Escalation Attacks on Android. In: Information Security. (2011)
32. de Raadt, T.: Exploit Mitigation Techniques (in OpenBSD, of course) (Nov. 2005)
33. Designer, S.: Getting around non-executable stack (and fix) (Aug. 1997)
34. Designer, S.: Linux kernel patch to remove stack exec permission (Apr. 1997)
35. Designer, S.: JPEG COM Marker Processing Vulnerability (Jul. 2000)
36. DilDog: L0pht Advisory MSIE4.0(1) (Jan. 1998)
37. Dowd, M.: Application-Specific Attacks: Leveraging the ActionScript Virtual Machine (Apr. 2008)
38. Durden, T.: Bypassing PaX ASLR Protection. Phrack Magazine (Jul. 2002)
39. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In: DIMVA. (Jul. 2009)
40. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks (Jun. 2000)
41. Fewer, S.: Pwn2Own 2011: IE8 on Windows 7 hijacked with 3 vulnerabilities (May 2011)
42. Fisher, D.: Survey Shows Most Flaws Sold For $5,000 Or Less (May 2010)
43. Fisher, D.: Chaouki Bekrar: The Man Behind the Bugs (Mar. 2012)
44. Fisher, D.: Offense is Being Pushed Underground (Mar. 2012)
45. Flake, H.: Third Generation Exploits. In: Blackhat USA Windows Security. (Feb. 2002)
46. Flake, H.: Exploitation and State Machines: Programming the "weird machine" revisited (Apr. 2011)
47. Fresi-Roglia, G., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: ACSAC. (Dec. 2009)
48. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In: Proceedings of the 21th USENIX conference on Security. (2012)
49. Goodin, D.: Legal goons threaten researcher for reporting security bug (2011)
50. Guido, D.: Vulnerability Disclosure (2011)
51. Hawkes, B.: Attacking the Vista Heap. In: Blackhat USA. (Aug. 2008)
52. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: Where'd My Gadgets Go? In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. (2012)
53. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: USENIX ATC. (2002)

54. Jones, R.W.M., Kelly, P.H.J., C, M., Errors, U.: Backwards-compatible bounds checking for arrays and pointers in c programs. In: Third International Workshop on Automated Debugging. (1997)
55. jp: Advanced Doug lea's malloc exploits. Phrack Magazine (Aug. 2003)
56. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization (Oct. 2003)
57. Kononenko, S.: Remote root vulnerability in Exim (Dec. 2010)
58. Krahmer, S.: x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique (Sept. 2005)
59. Labs, M.S.: Security Labs Report, July - December 2011 Recap (Feb. 2012)
60. Lemos, R.: Does Microsoft Need Bug Bounties? (May 2011)
61. Litchfield, D.: Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. In: Blackhat Asia. (Dec. 2003)
62. Litchfield, D.: Windows Heap Overflows. In: Blackhat USA Windows Security. (Jan. 2004)
63. Lopatic, T.: Vulnerability in NCSA HTTPD 1.3 (Feb. 1995)
64. Marinescu, A.: Windows Vista Heap Management Enhancements. In: Blackhat USA. (Aug. 2006)
65. MaXX: VUDO Malloc Tricks. Phrack Magazine (Aug. 2001)
66. McDonald, J.: Defeating Solaris/SPARC Non-Executable Stack Protection) (Mar. 1999)
67. McDonald, J., Valasek, C.: Practical Windows XP/2003 Heap Exploitation. In: Blackhat USA. (Jul. 2009)
68. Meer, H.: Memory Corruption Attacks The (almost) Complete History. In: Blackhat USA. (Jul. 2010)
69. Mein, A.: Celebrating one year of web vulnerability research (2012)
70. 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 (Sept. 2006)
71. Miller, M.: Preventing the Exploitation of SEH Overwrites (Sept. 2006)
72. Necula, G.C., Condit, J., Harren, M., Mcpeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy software. ACM Trans. on Progr. Lang. and Syst. (2005)
73. Nergal: The Advanced Return-Into-Lib(c) exploits (PaX Case study). Phrack Magazine (Dec. 2001)
74. NIST: The Second Static Analysis Tool Exposition (SATE) 2009. (Jun 2010)
75. Okun, V., Guthrie, W.F., Gaucher, R., Black, P.E.: Effect of static analysis tools on software security: preliminary investigation. In: Proceedings of the 2007 ACM workshop on Quality of protection. (2007)
76. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. In: ISSTA. (2002)
77. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. (2012)
78. Phantasmagoria, P.: The Malloc Maleficarum (Oct. 2005)
79. Planet, C.: A Eulogy for Format Strings. Phrack (Nov. 2010)
80. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: ACSAC. (2010)
81. Richarte, G.: Four different tricks to bypass StackShield and StackGuard protection (Jun. 2002)
82. Ruwase, O., Lam, M.: A practical dynamic buffer overflow detector. In: Proceedings of NDSS Symposium. (February 2004)

83. Ryan Roemer, Erik Buchanan, H.S., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM TISSEC (Apr 2010)
84. Salamat, B., Jackson, T., Gal, A., Franz., M.: Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space. In: EuroSys. (2009)
85. SANS: CWE/SANS TOP 25 Most Dangerous Software Errors (Jun. 2011)
86. Schmidt, C., Darby, T.: The What, Why, and How of the 1988 Internet Worm (Jul. 2001)
87. Scut: Exploiting Format String Vulnerabilities (Sept. 2001)
88. Seifried, K., Levy, E.: Interview with Elias Levy (Bugtraq) (2001)
89. Serna, F.J.: CVE-2012-0769, the case of the perfect info leak (Feb. 2012)
90. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the Effectiveness of Address-Space Randomization. In: ACM CCS. (Oct. 2004)
91. SkyLined: Internet Exploiter 3: Technical details (Nov. 2004)
92. SkyLined: Internet Explorer IFRAME src&name parameter BoF remote compromise (Oct. 2004)
93. SkyLined: Microsoft Internet Explorer DHTML Object handling vulnerabilities (MS05-20) (Apr. 2005)
94. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of NDSS 2011, San Diego, CA (2011)
95. Slowinska, A., Stancescu, T., Bos, H.: Body armor for binaries: preventing buffer overflows without recompilation. In: Proceedings of the USENIX Security Symposium. (2012)
96. StackShield: Stack Shield: A "stack smashing" technique protection tool for Linux (Dec. 1999)
97. Symantec: Symantec report on the underground economy (2008)
98. Team, P.: Address Space Layout Randomization (Mar. 2003)
99. The Pax Team: Design & Implementation of PAGEEXEC (2000)
100. Theriault, C.: Why is a 14-month-old patched Microsoft vulnerability still being exploited? (Feb. 2012)
101. Twillman, T.: Exploit for proftpd 1.2.0pre6 (Sept. 1999)
102. van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.: Memory Errors: The Past, the Present, and the Future. Technical Report IR-CS-73 (Nov. 2011)
103. Veracode: State of Software Security Report Volume 4. (Dec 2011)
104. VUPEN: Safari/MacBook first to fall at Pwn2Own 2011 (Mar. 2011)
105. VUPEN: Pwn2Own 2012: Google Chrome browser sandbox first to fall (Mar. 2012)
106. VUPEN: Pwn2Own 2012: IE 9 hacked with two 0day vulnerabilities (Mar. 2012)
107. Waisman, N.: Understanding and Bypassing Windows Heap Protection (Jun. 2007)
108. Wei, T., Wang, T., Duan, L., Luo, J.: Secure dynamic code generation against spraying. In: ACM CCS. (2010)
109. X-Force, I.: IBM X-Force 2011 Mid-year Trend and Risk Report. (Sep 2011)
110. Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical Report CW386 (Jul. 2004)
111. Younan, Y., Philippaerts, P., Cavallaro, L., Sekar, R., Piessens, F., Joosen, W.: PAriCheck: an efficient pointer arithmetic checker for c programs. In: AsiaCCS. (2010)
112. Zatko, P.: How to write Buffer Overflows (1995)
113. Zetter, K.: Three minutes with rain forrest puppy (2001)