On trends in low-level exploitation

Christian W. Otterstad

Department of Informatics, University of Bergen

Abstract

Low-level computer exploitation and its mitigation counterpart has accumulated some noteworthy history. Presently, especially in academia, it features a plethora of mitigation techniques and also various possible modes of attack. It has seen numerous developments building upon basic methods for both sides and certain trends have emerged. This paper is primarily an overview paper, focusing especially on x86 GNU/Linux. The basic reasons inherent for allowing low-level exploitability are identified and explained to provide background knowledge. The paper furthermore describes the history, present state of the art and future developments that are topical and appear to be important in the field. Several attack and defense techniques have overlapping notions with not always obvious differences. Herein the notion of the bar being raised for both exploits and mitigation methods is examined and extrapolated upon based on the known relevant present state and history. The difference between academia and the industry is discussed especially where it relates to application of new mitigation techniques. Based on this examination some patterns and trends are identified and a conjecture for the likely future development of both is presented and justified.

This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

1 Introduction and earlier related work

In 1972 the paper "Computer Security Technology Planning Study" was published [1]. Since then, research surrounding the main ideas in this paper has grown to become a mature and complex field in its own right. There are many different exploitation techniques and mitigation techniques. There are various issues associated with most of them and due to the sheer number and complexity it can be hard to evaluate the differences.

This paper does not attempt to provide an exhaustive list or to compare all existing exploitation techniques and mitigations. However, it does attempt to examine some of the noteworthy history, present developments in exploitation as a field, and make conjectures regarding its future development.

Since this paper is primarily an overview paper, other overview papers are the most relevant prior works.

"Memory Corruption Attacks The (almost) Complete History" [2] describes memory corruption attacks and mitigations up to 2010.

"Memory Errors: The Past, the Present, and the Future" [3] examines the history of low-level exploitation up to 2012 and also includes higher level exploitation techniques.

"SoK: Eternal War in Memory" from 2013 discusses some of the topics that are relevant to the current paper [4].

Certain papers also provide an overview of specific techniques that are relevant to their contribution [5] [6].

The contribution of this paper is to provide an overview taking into account the latest developments as of the writing date of the paper and provide a conjecture for likely future developments.

2 What is exploitation and what is the root cause of exploitability?

All exploits utilize existing functionality in the target program being exploited [7]. The exploited functionality is what is collectively called a bug, or more specifically an exploitable vulnerability. The notion of exploitability is therefore closely related to unintended functionality. As a corollary to the Halting problem, all functionality of a program cannot be determined programmatically. Furthermore, both manual code review and assisted debugging using dynamic or static tools can still overlook issues. Finding and correcting all bugs is an open problem in computer science and the problem of preventing all exploitable issues becomes akin to a subset of this problem. However, this is a simplified statement, since a bug does not necessarily need to be fixed to remove the ability to exploit it. Regardless, the observation justifies the statement that exploitable bugs are likely to remain an issue for the foreseeable future.

It should also be noted that exploitation can occur at multiple levels. At least in hardware, in software, and in the network, or a combination. Here, we are concerned with

low-level exploitation, which can be defined as unexpected computation resulting from a low-level vulnerability. It is important to note that the vulnerability may be beyond a developer's control, yet the software of that developer may still be affected by it. The vulnerability may reside in a third party library, or even in more underlying software or hardware utilized by the same software.

The attack scenario or attack model topical for this paper is very generic and simple. An attacker faces a system providing some form of interaction and the goal of the attacker is ideally arbitrary code execution at the highest privilege level of the system. If such an attack cannot be achieved, any control more powerful than the initial intended control is a potential benefit to the attacker.

All exploitation can be said to escalate privileges in some manner. The attacker can always interact with the target system, either directly or indirectly, and has as a goal to increase the influence or control over the system. The goal can also be seen as growing the set of operations the attacker can perform. If the attacker already has the privilege level required without an exploit, then there is no exploitation taking place but likely merely abuse of e.g. a misconfigured server. It should be noted that the terminology is not very precisely defined, but there are multiple possible types of exploit scenarios, regardless of the exploitaiton technique employed. In a local exploitation scenario the attacker already has an account on the system and attempts to escalate privileges. A scripting environment presents a different scenario where the attacker can run arbitrary code on the system as defined by the scripting language, e.g. JavaScript. In remote exploitation the attacker can only interact remotely with the target system without an existing account.

3 History

Detailed history has already been provided in the aforementioned "Memory Errors" paper by Veen, et al. [3] up to 2012. A selected summary to provide context will be given here with newer contributions mentioned where appropriate.

Computer exploitation issues have been reported at least as early as 1972 [1]. However, many years passed before the well-known hallmark papers appeared and it is not clear what published material may have preceded these works.

The 1988 Morris worm exploited i.a. a stack buffer overflow, but there was no paper or write-up associated with it. Later, an e-mail was published to Bugtraq regarding a vulnerability in NCSA HTTPD 1.3 with a proof-of-concept exploit implementation in C in the same write-up. However, it appears most papers cite Aleph1 as the primary source for the first stack based buffer overflow paper [8].

The natural reaction from the defender's standpoint to the stack overflow was W$\oplus$X (Write XOR Execute), to prevent the stack from being executable when not needed. Casper Dik and Solar Designer wrote patches for Solaris and Linux, respectively, to implement software based non-executable memory [9]. The PaX Team subsequently contributed to developing and extending non-executable memory support in Linux with their PAGEEXEC, SEGMEXEC and MPROTECT kernel patches from 2000. Later implementations would draw upon hardware support — the page granularity NX-bit (No-eXecute) — as it became available. This happened around 2003 and 2004 for Windows

and GNU/Linux respectively. Some programs require the stack to be executable so this feature can be toggled with mprotect(). Since the stack based buffer overflow has a heap based counterpart [10], the same mitigation is applied to the heap. Some complications had to be dealt with, e.g. that Linux requires an executable user space stack for signal processing and that function trampolines also require this [9]. The implemented solution was to allow executable stacks where needed. We see here the principle of not removing the attacker's functionality in all cases, but in many. The functionality that remains for the attacker becomes a subset of the previous functionality without the mitigation in place.

Return to libc (ret2libc) by Solar Designer in 1997 [11] introduced the notion of executing system library code instead of utilizing attacker supplied code. Non-executable pages do not prevent hijacking of the instruction pointer and thus the attacker is still free to execute any other executable memory. A year later in 1998, return address protection schemes (which should not be confused with the more modern RAP — Reuse Attack Protector) were introduced with an implementation called StackGuard [12]. This mitigation technique attempts to prevent the attacker from gaining control over the program counter in the first place. StackGuard saw attacks against it published in Phrack [13].

Format string based attacks and integer overflows also became more widely known around the year 2000 [14] [15] [16].

To defend against executing memory at known offsets, e.g. in the system library as with ret2libc, the PaX Team introduced ASLR (Address Space Layout Randomization) [17]. ASLR changes the nature of the attack from being deterministic to stochastic. Later, stronger versions of ASLR started appearing.

The PaX Team published a document "pax-future.txt" in 2003 [18], the concepts therein was later coined CFI (Control Flow Integrity) [19]. CFI conceptually attempts to only allow a whitelist of certain execution paths to occur for a given program.

It appears that 2005 saw the first published paper describing a purely data-oriented attack citeChen:2005:NAR:1251398.1251410, which is noteworthy as this idea is likely to be more topical again once attacker control is further constrained.

ROP (Return-Oriented programming) [20] from 2007 draws upon the notion introduced in ret2libc, namely that of code reuse, and generalizes it. No new code needs to be introduced by the attacker. The stack becomes a program counter, executing snippets of code — gadgets — all terminated by return instructions. The concatenation of such gadgets on the stack allows for Turing complete execution given a sufficiently large code base to glean gadgets from.

JOP (Jump-Oriented programming) [21] introduced in 2011 is an evolution of ROP, which precludes the need for a return oriented type of gadget. Jump-oriented branches may be utilized, coupled with a special dispatcher gadget which acts as a program counter. The dispatcher gadget maintains control by having the topical registers configured correctly. SOP (String-Oriented Programming) from 2013 [22] is another variant of code-reuse programming relying on the use of format strings.

The defense oriented papers responded to code reuse attacks by suggesting fine grained, high entropy ASLR implementations. Given that the attacker could leak a pointer, the relative offsets between memory regions would allow easy calculation of the offset at which the required code would be. However, with a fine grained ASLR implementations, such relative offsets could not be utilized in this manner.

2013 saw the first JIT-ROP (Just-In-Time Return-Oriented Programming) paper [23], which introduced the notion of an iterable information leak to build a full exploit. By iterating over a controllable information leak, the attacker can map an arbitrary region of memory. Hence, any permutation of code or shifting of offsets could be rediscovered by the attacker. The exploit could then fall back on traditional ROP with real-time computationally determined gadget offsets.

In 2014 the Hacking Blind paper [24] was published, describing how to exploit an unknown remote program in an automated manner. In the same year, the mitigation technique XnR (eXecute-no-Read) was published [25].

Kernel exploitation has also seen a development where the kernel is found to be vulnerable to many of the same issues prevalent in user space. However some issues are specific to the kernel. The kernel is mapped into the upper memory region of user space and could directly access arbitrary memory in user space due to performance demands. This — a technique known as ret2usr (Return-to-User) — allowed for mapping a region of memory in user space and writing shellcode to it, when dereferencing a pointer in kernel space pointing to this memory location the attacker would gain control over the control flow in kernel space. SMEP (Supervisor Mode Execution Prevention) and SMAP (Supervisor Mode Access Prevention) were introduced in 2012. Enabled by setting a bit for each mitigation in the CR4 register, these techniques prevent the kernel from executing and accessing user space memory, respectively. In 2015 the ret2dir technique by Vasileios Kemerlis kernel exploitation technique was introduced, bypassing SMEP and SMAP [26].

2015 saw the introduction of MPX (Memory Protection Extensions), a hardware implementation of concepts earlier used in software e.g. in AddressSanitizer. MPX offers bounds checking in instrumented binaries with hardware accelerated instructions specifically made for that task. Such techniques require recompilation and also incur a performance penalty.

RAP (Reuse Attack Protector) was introduced in 2015 by the PaX Team. [27] RAP offers CFI-like protection whilst tackling the issue of too coarse grained and/or too performance demanding implementations.

4 ASLR

ASLR may sometimes be confused with ASLP and vice-versa. Originally, ASLR only shifted the offset of where a region of memory started and provided some entropy for this expressed in bits. ASLP will permute the contents of the region of memory itself, which may also be expressed as a certain number of bits of entropy.

Some general challenges are to utilize the full address space as an offset, the kernel does not readily allow for this. Furthermore, the ASLR implementation should permute

the contents at the highest granularity. This requires recompilation or rewriting the binary, which often does not scale well to large programs. Finally there is the requirement of not introducing a high performance penalty and avoid compatibility or usability issues.

Some notable ASLR/ASLP proposals are now mentioned.

- 2012: ORP Performs ASLP by static rewriting of basic blocks. No shifting of offsets is performed but the use of instructions and registers is randomized [5] [28].
- 2012: STIR [29] performs ASLP-like permutation and randomizes the base address. It claims a performance overhead of 1.6% and file size increases by 73%.
- 2012: ILR [30] cannot resolve all indirect branches but claims to be able to able to conceptually permute every instruction in a program.
- 2012: Fiuffrida, et al. [31] published a paper about kernel space ASLR. An implementation for Linux called KASLR has since been available for the standard Linux kernel, however it has low entropy and can be defeated by information leaks.
- 2013: XIFER [5], XIFER claims a runtime overhead of 1.2%. It appears to offer $\log_2(16!) \approx 44.25$ bits of entropy.
- 2016: ASLR-NG [32] provides a variable ASLR entropy which appears to be superior to both the standard Linux and PaX ASLR implementations. It provides 43 bits for the stack and heap and 35 bits for the executable.
- 2016: Selfrando [33] performs ASLP-like function permutation. It has a total entropy of $E_t = log_2(m!)$ [33], where m is the number of functions in the code that is permuted.

Some strong ASLR implementations, such as STIR, may be vulnerable to code generated on the fly by the target program, e.g. a virtual machine or emulator. An attack scenario of this type may also be vulnerable to classic code injection. The common issue all these mitigations have is that they are all vulnerable to JIT-ROP [23]. However, in cases where there is no iterable memory leak, or the number of iterations is limited in number or speed they are likely to provide a more effective defense. However, even then, data-oriented attacks may pose a threat, depending on the nature of the system and vulnerability. It has also been shown that ASLR may be defeated locally by exploiting timing attacks based on the BTB (Branch Target Buffer) as long as basic blocks are not permuted [34].

5 XnR

JIT-ROP [23] prompted the introduction of XnR. Several proposals exist, with some differences: XnR [25], HideM [35], Heisenbyte [36], and NEAR. [37].

While XnR-like techniques prevent an attacker from reading certain pages of memory, there is no protection against execution of the same memory. Fine-grained XnR implementations with sub-page granularity to distinguish mixed code and data pages would have the same problem. The Hacking Blind paper [24] has shown that memory can be indirectly read by mere execution of memory. However, this requires a restarting

server model which does not re-randomize. The scanning time required is also highly dependent on the strength of the ASLR implementation that is coupled with the XnR implementation. It has also been found that XnR can be defeated when using JIT-ROP in a scripting environment on Windows [37].

6 CFI

The formalized notion of control flow started with the paper from Abadi, et al. in 2005 [19] although as previously mentioned PaX suggested the notion already in 2003 [18]. In 2006, a suggestion for DFI (Data-Flow Integrity) was published by Microsoft, however its coverage is not perfect [38]. Some noteworthy implementations are: kBouncer, ROPecker, CFI for COTS (Commercial off-the-shelf) binaries, ROP-Guard, Microsoft EMET (Enhanced Mitigation Experience Toolkit), [6], CET (Control-flow Enforcement Technology), and RAP.

A common issue with several CFI implementations is being too coarse-grained, as pointed out in [6] which demonstrate Turing-complete ROP attacks in such coarse cases. The Control-flow bending paper by Carlini, et al. [39] argues that shadow call stacks appear to be essential for the security of CFI. Dang, et al. [40] argues that shadow call stacks have a lower bound of 3.5% overhead, which enforces a minimal performance cost overhead on effective CFI for standard x86. However, hardware based solutions claim 0.5% to 1.75% overhead, depending on the benchmark used [41].

Some hardware acceleration has emerged in the form of using performance counters to implement CFI [42]. However, CFI appears to have an upper bound related to the halting problem, as previously pointed out [18]. Intel has also suggested a hardware based system CET but with no actual hardware to run it as of this writing date [43]. The PaX Team has criticized CET, suggesting that i.a. its indirect branch tracking allows too broad target ranges [44] something which RAP appears vastly superior at. The commercial version of RAP seems to offer protection of all return statements as well as all calls [27].

7 Data-oriented attacks

The first paper describing data-oriented attacks in a formalized manner appears to be Chen, et al. [45]. An automatic approach to generating exploits based on this technique was published in 2015 [46] demonstrating a prototype tool called FLOWSTITCH that could automatically construct exploits based on data-oriented attacks.

The ability to construct advanced data-oriented attacks using only legal control flow paths appears to be of considerable importance if assuming that attacks targeting the control flow will become deprecated [27].

8 Other mitigations

Any mitigation that constrains the set of operations the attacker can perform would be beneficial for the defender in the sense of enforcing the principle of least privilege. It would be advantageous to, e.g., use RBAC (Role-Based Access Control) to reduce the set of operations a possible vector of attack may utilize as compared to DAC (Discretionary Access Control). Virtual machines and containers also restrict the functionality

offered to the attacker and can be cost-effective ways to improve security. However, they are all vulnerable to exploits, like any other software. CVE-2014-9357, CVE-2009-1244, CVE-2014-0983, CVE-2015-3456 are examples of vulnerabilities for Docker, VMware, Virtualbox, and QEMU, respectively.

Another possible mitigation is microservices. Microservices, utilized in the correct manner, can possibly help mitigate general issues with exploits, as also discussed in [47]. Especially when coupled with strong ASLR/ASLP with binary rewriting, an attacker facing a microservice oriented architecture will in some cases only obtain a subset of the control traditionally obtained over a monolithic system. The attacker would then have to utilize additional exploits to move laterally within the microservice network from the compromised service in order to gain the control flow on other services. The attacker may also remain in the subset of nodes already under control and attempt to issue higher level commands to the system, which may or may not be sufficient. Either way it appears to constrain the influence the attacker has and/or make it more costly. Whereas with a monolithic program, the attacker would control the whole program once having successfully hijacked the instruction pointer.

9 Requirements for adopting a mitigation technique in the industry

Herein we denote the industry as non-academia and non-security experts. We note that few mitigation techniques are adopted by the industry.

Interestingly, it appears the industry dictates what security mitigations are actually incorporated, and when. In some cases, as pointed out in Section 3 it may take years before effective and tested mitigation techniques are incorporated. On the other hand there is no authoritative body within security research which strongly attempts to dictate what security mitigations should be incorporated. The industry sometimes has good reasons for being reluctant to incorporating security features. Some criteria that are usually evaluated have been suggested in [4]. These are protection (enforced policy, false negatives, false positives); cost (performance overhead, memory overhead), as also pointed out by [48]; and compatibility (source compatibility, binary compatibility, modularity support).

However, there does not appear to be any way to formally evaluate or quantify these attributes. E.g. in the sense what percentage of overhead is acceptable performance loss, and in what situations? There are many complicating factors; certain mitigations may only have an average overhead, with outliers in specific situations. Consider that a paper [40] has shown that it is unlikely the performance overhead cost of a shadow stack can be brought below 3.5%. If this lower bound is still too high, it would be useful for academia to know this beforehand, since resources could then likely be better spent going in other directions. It has been suggested that an overhead larger than 10% usually does not get adopted whereas others suggest 5%[4].

The industry does not seem to maintain any formalized framework suited for evaluating security mitigations. While generalized risk evaluations exist they are not suited for evaluating complex exploit mitigation techniques. As an example, Windows got support for ASLR and W$\oplus$X 4 and 5 years respectively after the PaX Team introduced Linux patches for these features. While there are costs involved with introducing such tech-

niques, such as development costs, the reasons for this exact latency appear to be largely arbitrary.

Gibbon's paper "Science's new social contract with society" suggests that the authority of science must be legitimated again and again [49]. However, considering the present state of open problems with how to improve security, it seems that allowing the industry to dictate — as in legitimating the authority of science — only further complicates the issue. Especially if such evaluations are based on largely arbitrary grounds. It would seem that a better set of criteria that is more strictly adhered to would allow more readily the adoption of security mitigation features.

10 The future of exploitation

Ideally, the defender does not want to allow any operation that is not strictly required by the program to offer its intended functionality. However, the granularity at which such operations are allowed is not arbitrary. On the x86, e.g. memory permissions are enforced at page level granularity, whereas computational operations are constrained at a much higher granularity, such as typically ring 3 and ring 0. While it is possible to introduce higher granularity by software, this often comes at a cost. Looking at the history and present day reluctance for adoption of new mitigation techniques, it does not appear likely that even modest performance overhead will be accepted. Therefore, it seems fair to assume that the granularity will only be increased on most machines in the industry after the overhead approximates close to zero, either with very efficient software implementations or hardware implementations. Even then, based on history, it might take time before actual adoptions occur.

The ideal — but unrealistic — situation for the defender would be to enforce the minimal principle of least privilege at the hardware level and only allow a certain set of machine states for the entire duration of the program, i.e. arbitrarily high precision CFI/DFI. The entire state of the machine would be verified to be correct per operation the system performs. Since this would require storing not only the registers but also all of the RAM — basically the whole machine state — even with optimizations this does not appear feasible. It also appears to introduce the halting problem again.

For now, the most open problem appears to be exploitation attempts that only use legal control flow paths and use data-oriented attacks [27].

The installation of a rootkit should be distinguished from the exploitation itself. Traditionally, the installation would only require root access, both when installing a user mode based rootkit and a kernel based one. However, modern systems may protect the system in various ways, such as forcing the attacker to deal with BIOS_CNT, PRx, and Boot Guard [50].

A consequence of modern defences is that the lines between exploitation and rootkit installation have become blurred. Installing a rootkit on a system that requires modules to be signed, employs DRTM (Dynamic Root of Trust Measurement) or SRTM (Static Root of Trust Measurement), may require an exploit or other approaches. The same may hold true for placing malicious code in SMM, the BIOS, in a DMA capable device or in the microcode. Whenever rootkit operations require low-level exploitation they naturally fall

into the domain of exploitation.

A possible consequence of highly sophisticated mitigation systems is that not only are exploits still possible in some cases, but backdoors disguised as bugs are still a possibility even when many combinations of rare cases would be required. A backdoor can be implemented as an exploit for a deliberate bug which may be hard to differentiate from a non-intentional bug. For this reason such an approach to writing backdoors appear to readily allow for repudiation. It is unclear if the presence of multiple advanced mitigation systems makes this attack vector more or less favorable but it seems fair to suggest it would still be a possibility.

A clear trend, which has been pointed out before [4], appears to be that the the industry is reluctant to employ mitigation techniques that incur a significant cost. This may not always be the optimal choice — even in purely maximizing profit — as the cost of problems later may be difficult to calculate, especially if considering black swan type of problems. However, complexity is increasing while the control offered by success may decrease. Extrapolating somewhat on this, it seems a fair conjecture to make that it will become increasingly rare — but certainly still possible — to see single individuals developing complex exploits. More often than before, it may be required to have teams with different skill-sets to make a project feasible in some cases and the resources available to the team become more topical. An eventuality of this could further favor government sponsored teams that can take more risk and have more resources at their disposal.

11 Conclusion

Building upon previous statements, it has been argued that the level of control granted from an exploitable bug is diminishing and that the techniques required to do so are getting increasingly complex. Furthermore, it appears the likely development and adoption of modern mitigation techniques will eventually reduce the possible exploitation techniques to data-oriented exploits in many cases. It has also been suggested that hardware accelerated features may further limit the amount of control wielded by a data-oriented exploit. For now, there seems to be a clear trade-off between the granularity at which attacks may be detected and the cost to do so. As this cost comes down the detection rate and constrained influence the attacker has are likely to be positively affected for the defender.

References

[1] J. P. Anderson, "Computer Security technology planning study." `http://csrc.nist.gov/publications/history/ande72.pdf`, 1972. [Online; accessed 24-August-2016].

[2] H. Meer, "Memory corruption attacks the (almost) complete history." Black Hat USA, August 2010, August 2010. [Online; accessed 19-October-2016].

[3] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, (Berlin, Heidelberg), pp. 86–106, Springer-Verlag, 2012.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 48–62, IEEE Computer Society, 2013.

[5] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), pp. 299–310, ACM, 2013.

[6] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 401–416, USENIX Association, 2014.

[7] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to weird machines and theory of computation." `http://www.cs.dartmouth.edu/~sergey/langsec/papers/Bratus.pdf`, December 2011.

[8] Aleph1, "Smashing the stack for fun and profit." PHRACK Magazine, vol. 7, no. 49, file 14 of 16, 1996.

[9] C. Yang-Seo, S. Dong-il, and S. Sung-Won, *A New Stack Buffer Overflow Hacking Defense Technique with Memory Address Confirmation*, pp. 146–159. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[10] anonymous, "Once upon a free()...." Phrack Inc, Volume 0x0b, Issue 0x39, Phile 0x09 of 0x12, 2001. [Online; accessed 23-August-2016].

[11] Solar Designer, "Getting around non-executable stack (and fix)." `http://seclists.org/bugtraq/1997/Aug/63`, August 1997. [Online; accessed 24-August-2016].

[12] C. Cowan, C. Pu, D. Maier, H. Hintony, 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 Conference on USENIX Security Symposium - Volume 7*, SSYM'98, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 1998.

[13] Bulba and Kil3r, "Bypassing stackguard and stackshield." PHRACK Magazine, vol. 10, no. 56, file 5 of 16. [Online; accessed 24-August-2016].

[14] Scut at Team Teso, "Exploiting format string vulnerabilities, version 1.2." `https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf`, March 2001. [Online; accessed 23-August-2016].

[15] gera and riq, "Advances in format string exploitation." PHRACK Magazine, vol. 11, no. 59, file 7 of 18. [Online; accessed 24-August-2016].

[16] blexim, "Phrack inc, volume 0x0b, issue 0x3c, phile 0x0a of 0x10." `http://phrack.org/issues/60/10.html`, December 2002. [Online; accessed 24-August-2016].

[17] PaX Team, "aslr.txt." `http://pax.grsecurity.net/docs/aslr.txt`, March 2003. [Online; accessed 24-August-2016].

[18] PaX Team, "pax-future.txt." `https://pax.grsecurity.net/docs/pax-future.txt`, March 2003. [Online; accessed 24-August-2016].

[19] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, (New York, NY, USA), pp. 340–353, ACM, 2005.

[20] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 552–561, ACM, 2007.

[21] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), pp. 30–40, ACM, 2011.

[22] M. Payer and T. R. Gross, "String oriented programming: when aslr is not enough," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, (New York, NY, USA), pp. 2:1–2:9, ACM, 2013.

[23] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 574–588, IEEE Computer Society, 2013.

[24] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 227–242, IEEE Computer Society, 2014.

[25] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 1342–1353, ACM, 2014.

[26] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 957–972, USENIX Association, Aug. 2014.

[27] PaX Team, "Rap: Rip rop." `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, October 2015. [Online; accessed 24-August-2016].

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, (Washington, DC, USA), pp. 601–615, IEEE Computer Society, 2012.

[29] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 157–168, ACM, 2012.

[30] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "Ilr: Where'd my gadgets go?," in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 571–585, May 2012.

[31] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, (Berkeley, CA, USA), pp. 40–40, USENIX Association, 2012.

[32] H. Marco and I. Ripoll, "ASLR-NG: ASLR Next Generation." `http://cybersecurity.upv.es/solutions/aslr-ng/aslr-ng.html`, April 2016. [Online; accessed 25-August-2016].

[33] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *The annual Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[34] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr." `http://www.cs.ucr.edu/~nael/pubs/micro16.pdf`, October 2016. [Online; accessed 20-October-2016].

[35] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, (New York, NY, USA), pp. 325–336, ACM, 2015.

[36] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 256–267, ACM, 2015.

[37] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, (New York, NY, USA), pp. 35–46, ACM, 2016.

[38] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 147–160, USENIX Association, 2006.

[39] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 161–176, USENIX Association, 2015.

[40] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, (New York, NY, USA), pp. 555–566, ACM, 2015.

[41] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, (New York, NY, USA), pp. 163:1–163:6, ACM, 2016.

[42] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[43] "Intel®." `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, June 2016. [Online; accessed 30-August-2016].

[44] "Close, but no cigar: On the effectiveness of intel's cet against code reuse attacks." `https://forums.grsecurity.net/viewtopic.php?f=7&t=4490&sid=423204cbf3f9297d44eb975533e038ea`, June 2016. [Online; accessed 30-August-2016].

[45] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2005.

[46] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, (Berkeley, CA, USA), pp. 177–192, USENIX Association, 2015.

[47] O. Lysne, K. J. Hole, C. Otterstad, Ø. Ytrehus, R. Aarseth, and J. Tellnes, "Vendor malware: Detection limits and mitigation," *IEEE Computer*, 2016.

[48] PaX Team, "Frequently asked questions about rap." `https://grsecurity.net/rap_faq.php`, 2016. [Online; accessed 24-August-2016].

[49] M. Gibbons *Nature*, vol. 402, pp. C81–C84, dec 1999.

[50] D. Oleksiuk, "Exploring and exploiting lenovo firmware secrets." `http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html`, June 2016. [Online; accessed 25-August-2016].