The Future of Buffer Overflows

Nicholas Lemonias, M.Sc. Information Security

The University of Derby, United Kingdom

December, 2012. Advanced Concepts in Secure Systems (Kim Smith)

Abstract — **Buffer Overflows are conditions relating to the Von-Neumann orientated model of computer architectures. Buffer Overflow conditions have been known for the past 25 years, and are still very common today and still ranked in the top threats of the threat landscape. The aim of this essay is to identify the attack vectors, the offensive and also defensive methodologies in conjunction to a chronological evaluation of the first, second and third generation buffer overflows. The objective is to critically evaluate the past, present and foreseeable future of buffer overflow conditions; The objective of this essay is to analyze buffer overflow conditions, and thus to study the reasons for their occurrence, to investigate the cause and effect of their occurrence in modern computer systems, and the predicament they create; The question to be raised "Why do buffer overflows continue to exist today, even with safer operating systems and programming languages?" Challenging the drawbacks of conventional approaches an evaluation of security vulnerabilities and modern attack methodologies enable us to speculate, on the foreseeable future of memory exploitation.**

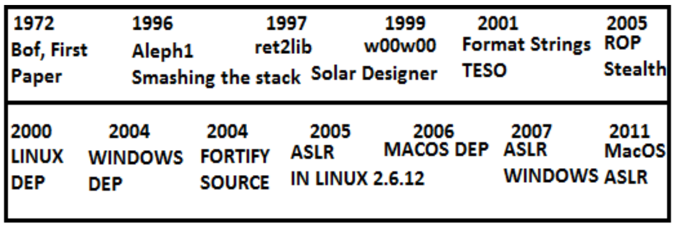
1. Introduction

In programming languages such as C and C++, the programmer is responsible for the adequate provision of manual memory management metrics and security controls in the security life cycle of a development tool, and since languages such as C and C++ have no inherent built-in protections for memory management, and this has to be provided by the programmer. Thus if the programmer is not careful to construct the security development lifecycle, for managing memory issues manually, then there is a great risk that hundreds of lines of legacy code could be unconditionally exposed to memory errors, and therefore exposing the system to a variety of vulnerabilities. Buffer overflow conditions occur when a fixed-length variable, stored either locally, or in the dynamic allocated segments of memory, is overflowed with excessive information. In such instances the excess data would overwrite the program's memory, the referenced buffer and adjacent memory locations next to it. A malicious adversary could take advantage, of the unprotected memory space thus with intent to circumvent the programs execution flow, and to directly tamper with the program's memory, and thus to execute malicious code in other unprotected adjacent executable locations in memory. Memory error exploitation is also dependent on other co-factors such as the microprocessor computer architecture (CPU enforces some basic constraints on memory access and can set control between privilege levels), can set control of the Operating System, the access control model (e.g.: RBAC, DAC), or the system's TCB (Trusted Computing Base – which contains the assurance mechanisms entailing also the access control to memory locations), and the reference monitor which regulates access of subjects to objects. A buffer overflow condition occurs when the software makes use of a subroutine and within that subroutine call, the user is allowed either by software design, or by poor programming practice, to input 'excessive' data into a fixed-length buffer, greater than the actual size it can store. A buffer overflow attack can further be identified as either local or remote. In a local overflow, the attacker aims for '***Escalation of Privileges'*** or (***EoP***)' on a system that he already has user-access on. In a remote overflow attack the malicious adversary overflows an active service by placing a malicious payload in the transmission data, with ultimate purpose to crash the service, or to cause an exception, and finally to trigger a buffer overflow condition, and subsequently control of the program's execution flow. This would eventually lead to access control at ring-0 or escalation of privileges with further exploitations, from ring-3 (user-land) to ring-0 (kernel-land); Classification of buffer overflows occurs according to their chronological discovery and methodology.

2. The Evolution of Buffer overflows

The emergence of 'Buffer Overflow' conditions dates back to the early 1970's with the Morris worm making its debut. The Internet worm that nearly shut down the Internet of that time, and of course took advantage of buffer overflow conditions, authored by a Cornell University student - Robert Tappan Morris. The Morris worm was a good enough cause for DARPA to found the establishment of CERT (Computer Emergency Response Team), at Carnegie Mellon University, and furthermore was a good enough cause for the emergence of the first 'Internet Law' the "US, Computer Fraud and Abuse patent". Robert Tappan Morris, the motivation of freedom of information inspired many security enthusiasts to author about buffer overflow conditions, and such are "Elias Levy" or also known as "Aleph 1", publishing the first ever public article on Buffer Overflows, "Smashing the Stack for Fun and Profit". Because of the lack of literature on memory exploitation trends then, in November, 1993 Scott Chasin founded the first ever white hat hacker mailing-list "Bugtraq", and generally the lack of a security community at that time. Furthermore, in 1995, Thomas Lopatic, published a step-by-step exploitation tutorial on the “NCSA HTTP” service. Peiter Zatko or also known in the hacker-scene by the alias “Mudge” has authored an article on stack-based overflows months after Lopatic’s publication. The introduction of a non-executable stack (NX Stack) was the first ever countermeasure to address memory exploitation, and that was authored by another famous name in the hacker-scene “Alexander Peslyak” – also known as “Solar Designer”- . Alexander was the first to publish a software implementation of the non-executable stack called “Stack Patch”, and that came in 1997. Crispin Cowan introduced SSP and Canaries in 1998. In 1999 the w00w00 team, a group of white-hat security enthusiasts were the first to author about heap overflows. In September, 1999 Tymm Twillman” introduced the first literature on format string attacks following a security audit of the ProFtpd daemon, in which the audit uncovered an insecure snprintf() call that directly passed user-generated data. Tillman’s journal gained so much popularity even years after his ProFtpd publication. Despite 25 years independent and academic research, buffer overflow conditions continue to exist, and the research community has introduced numerous innovative approaches for the effective mitigation of memory vulnerabilities. Such approaches incorporate the utilization of software for automated ‘bound-checking’, safer compiler options, kernel modules, structure handling, novel techniques such as probabilistic and entropy based randomization, and virtualization, secure sandboxing techniques. The first ever defensive ‘non-executable’ stack mechanism, against Buffer overflows was proposed by ‘Alexander Peslyak’ in 1997 along with a proof of concept exploit of ‘retun-to-libc’ attacks. The now popular “return-to- libc” attacks can still be used to defeat a non-executable stack. In 2000, the PaX project was released.

In July 2001, The PaX team also introduced the concept of a new technology “Address Space Layout Randomization” (ASLR). The concept elucidated that randomization of executable locations, and randomization of memory addresses would act as an adequate preventive barrier against code-reuse, and injection attacks. According to subject literature (Negral, 2001), some of the first custom attacks on randomization that remain a popular attack vector in present day, are the return-to-libc attacks. In 2008, Mark Dowd presented the first ever publication on NULL point de-references. Buffer overflow conditions continue to thrive and are still classified as one of the top threats, in the current threat landscape. However, what changed since, in terms of the protection mechanisms, are the enhanced mitigation techniques that contributed to novel defensive techniques, which constituted exploitation to become tougher, than previous controversial methodologies. The main problem is that vulnerabilities are profligate, and hundreds of programs run with root privileges by default.



1972 Bof, First Paper	1996 Aleph1 Smashing the stack	1997 ret2lib	1999 w00w00 Solar Designer	2001 Format Strings TESO	2005 ROP Stealth	
2000 LINUX DEP	2004 WINDOWS DEP	2004 FORTIFY SOURCE	2005 ASLR IN LINUX 2.6.12	2006 MACOS DEP	2007 ASLR WINDOWS ASLR	2011 MacOS

Fig. 1. Chronological evolution of Buffer Overflows.

3. Trend Analysis and Evaluation

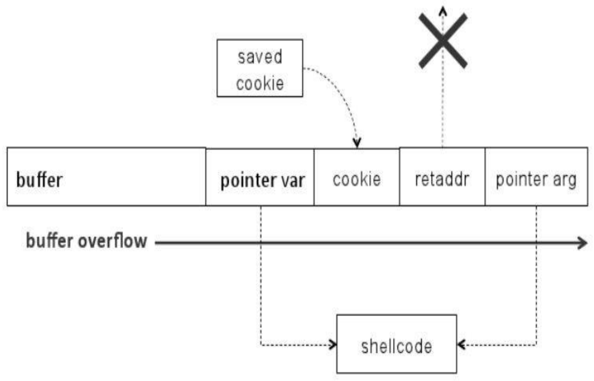
3.1 *W $\oplus$ X (Data Execution Prevention)*

Write **XOR** Execute is an orthogonal methodology developed for the protection against injection attacks, which is evident in first and second generation buffer overflows. Subject matter literature suggests that, first generation overflows take occurrence due to the feasibility to overwrite fixed-space addresses, a condition which is referred to as a ‘Fandango Core’. The W $\oplus$ X mechanism (also referred to as Data Execution Prevention – DEP) accumulates the micro-processor to mark adjacent memory as non-executable, in areas such as the heap, the program’s stack and the mmap section – (shared libraries), as well as the ELF section in the program header table. Thus the following defines the W $\oplus$ X concept: “Writable memory pages, but not executable,” or either “Executable, but then not writable, but not both.” Based on this concept malicious adversaries are barred from making changes to memory, by overwriting tables as in classical stack-overflows. A threat-actor at this point, would only be given the option to abuse what is already existent in memory, and such are function calls to existing libraries. However, DEP as a mitigation method has its efficacy rated high, especially due to its effectiveness to guard against stack-overflows. DEP is considered the best non-executable defence against stack overflows. DEP is often complemented with ASLR to enhance its efficacy. However, this mitigation technique has been effectively defeated using a (1) ‘return-to-libc’ attack (2) Return Object Orientated Programming - ROP attack – which can disable DEP/NX, (3) “NOPSled” attack. (Zheng Xu, 2011).

3.2 *Stack Smashing Protection (SSP)*

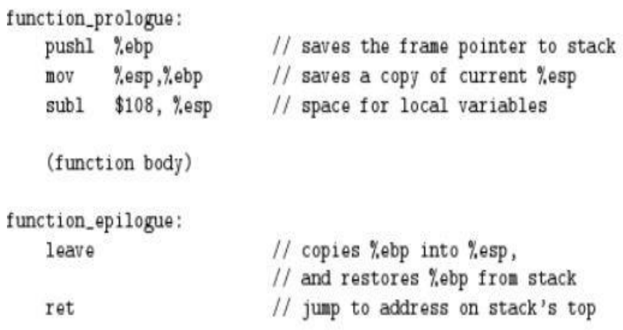
SSP is a mitigation method and it can be applied at compile-time. This mitigation method works by altering every function’s prologue and epilogue regions. This mitigation methodology inserts a copy of a ‘detection cookie’ in a program’s stack, just right before the program’s return address. Therefore if a buffer overflow was to occur within a function, the cookie would have to get overwritten first. In case it gets overwritten, an exception would be thrown. This mechanism works by comparison of the Stack cookie and a wider master cookie which is located in the .data section.
It is pertinent to note, that an exception is a critical error which defines that the program should be terminated and

therefore has to step outside its normal execution flow. This method is an effective mitigation, specifically for stack-based overflows, but its efficacy has been rated as very low since it cannot protect against heap overflows, and allocated structures. This method can be defeated either by (1) Go Beyond the EIP register overwriting the exception handler, using a Structured Exception Handler (SEH) exploit and redirecting that to a shell code. (2) Overwrite parent calls of object and V-Table pointers inside the stack layout (3) Guess where the cookie is and reduce its entropy. Although SSP significantly reduces the probability of a single attempt succeeding, it will however not prevent exploitation. Although it can reduce the impact of the attack. If it is combined with DEP then a return-to-libc / "stack-pivot" or using a Return-Object Orientated Programming – ROP attack". (Cowan, 1991)



/

Fig. 2. SSP and the relationship to the Stack Layout.



```
function_prologue:
    pushl  %ebp              // saves the frame pointer to stack
    mov    %esp,%ebp         // saves a copy of current %esp
    subl   $108, %esp        // space for local variables

    (function body)

function_epilogue:
    leave                    // copies %ebp into %esp,
                             // and restores %ebp from stack
    ret                      // jump to address on stack's top
```

Fig. 3. Stack Smashing Protection epilogue and prologue sections. (Gerardo, 2002).

3.3 *Address Space Layout Randomization*

The first, second and third generation of buffer overflows and general attacks against non-executable stack methodologies were reliant on knowledge of fixed memory addresses. In the virtual address space each page has its 'virtual addresses assigned to a physical memory location through page tables. Classical "return-to-libc" attacks required knowledge of the exact offsets in memory, normally referenced to the '**libc'** function which is used by all C orientated operating systems. Thus ASLR reduces the impact of a buffer overflow attack, by randomizing all base addresses in "The Heap", "The shared libraries **mmap()** area", "Code segment" and parts of an ELF (executable) binary. The ASLR pseudorandom algorithm randomizes only the first bits of a virtual address. In fact the 16 bits out of the 32-bits are randomized (32bits or 64 architectures) using a PRNG pseudo random number generator.

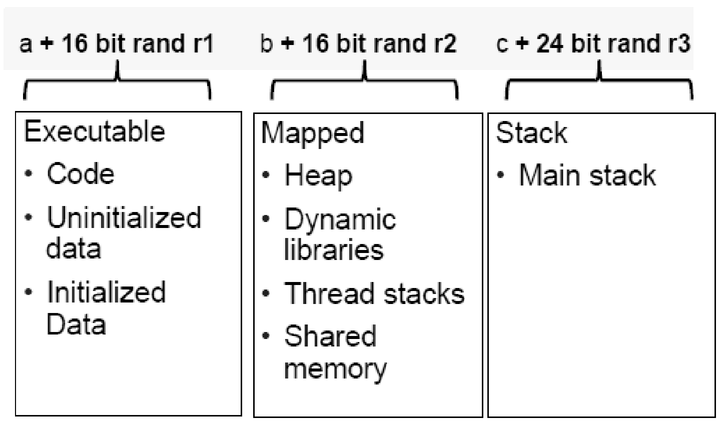


Fig. 4. Address Space Layout Randomization and base addresses. (Brumley, 2010)

The space of randomization is only 2^{16} on a 32 bit architecture, thus giving an attacker the probability of success with:

$$\frac{1}{2^{16}} = \text{Probability}$$

(1)

$$\underbrace{\frac{2^n-1}{2^n}\cdot\frac{2^n-2}{2^n-1}\cdots\frac{2^n-t-1}{2^n-t}}_{\Pr[\text{first } t-1 \text{ probes fail}]}\cdot\frac{1}{2^n-t-1}=\frac{1}{2^n},$$

(2)

Furthermore on the probability of success with brute forcing in case that a malicious adversary brute forces the "fixed" addresses of the randomized space, the expected number of probes required by a brute force attack can only be 2^{16} , while re-randomization of the address space can be thought of as selecting replaceable balls, to the point we get the ball with the number we are looking for. Therefore the probability of success after a number of probes "t", and thus where "n" is defined as the number of randomized bits in the address space - the success rate would be equivalent to the following equation below, which defines the expected number of attempts before a successful probe occurs, which is defined as 1 divided by 2 to the power of "n" probes, and which is therefore equal to 2^"n" bits plus 1 divided by 2. According to current literature, re-randomization occurs constantly for all child processes, and periodically after every call to the fork function, re-randomizing offsets of all child processes.

$$\sum_{t=1}^{2^n} t \cdot \frac{1}{2^n} = \frac{1}{2^n} \cdot \sum_{t=1}^{2^n} t = (2^n + 1)/2 \approx 2^{n-1}.$$

(3) Probability expectations

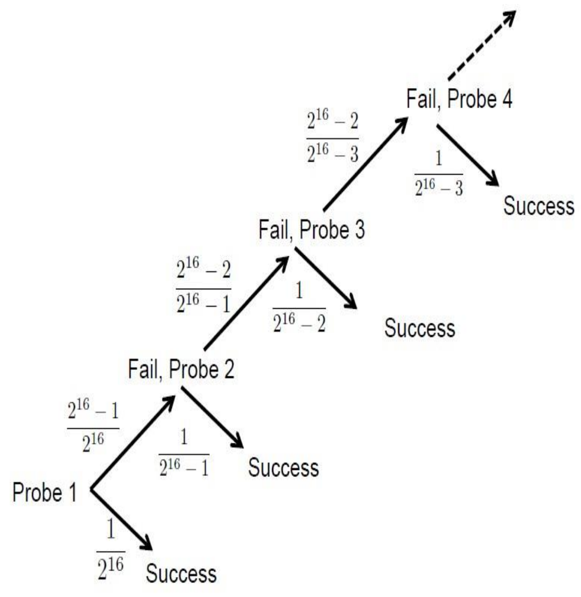


Fig. 5. Estimated relationship between the successful probes and the randomization bits utilized in ASLR. (Brumley, 2010)

$$\underbrace{\frac{2^{16}-1}{2^{16}} * \frac{2^{16}-2}{2^{16}-1} * \dots * \frac{2^{16}-n-1}{2^{16}-n}}_{\text{Fail the first n-1 times}} * \underset{\substack{\uparrow \\ \text{Succeed on nth trial}}}{\frac{1}{2^{16}-n-1}} = \frac{1}{2^{16}}$$

Trial and Error - Blind Bruteforcing attack methodology.

Fig. 6. Estimated relationship between the number of tries and randomization bits used in ASLR. (Brumley, 2010)

Address Space Layout Randomization is susceptible to brute-force attacks, partial-overwrites, information leak attacks (e.g: **printf()** arbitrary read or vulnerabilities that disclose information about the memory structure), and JIT spraying attacks. The success of brute-forcing an ASLR exploit is dependent on how tolerant an exploit is to the memory layout. Address Space Layout Randomization is often complemented with DEP or other mitigation technologies, due to deficiencies in its design that limit its efficacy and mitigation capabilities. ASLR can be bypassed with a partial EIP overwrite due to the fact that the least significant non-executable bits are not used. The following methodologies have successfully defeated ASLR; (Schacham, 2004), (Schwartz, 2012) (Sotirov, 2010)

1) Return-to-Libc attacks. (Calls outside the stack function)
2) Heap Spraying.
3) Information Leak attacks.
4) Brute forcing.
5) Return-Object-Orientated-Programming – ROP attack. (similar to return-to-libc but uses gadgets)
6) Just-in-Time (JIT) Spraying (JavaScript Payload)

3.4 Fine-Grained User-Space Security through Virtualization

Programs in a nutshell are verified, with additional security barriers added to the executable image. User-space virtualization confines and secures the application. The code is secured and system calls are authorized. All insecure functions are protected or rewritten.

i. Secure control flow transfers. (Verification of return addresses on the stack and protection from Return Orientated Programming, code injections through heap and stack-based overflows)
ii. Signal handling. (Protects from break-outs out of the sandbox)
iii. Executable bit removal. (Executable bit removal for libraries and applications).
iv. Address Space Layout Randomization. (Probabilistic Measures that makes attacks more difficult to circumvent).
v. Protection of internal data-structures.

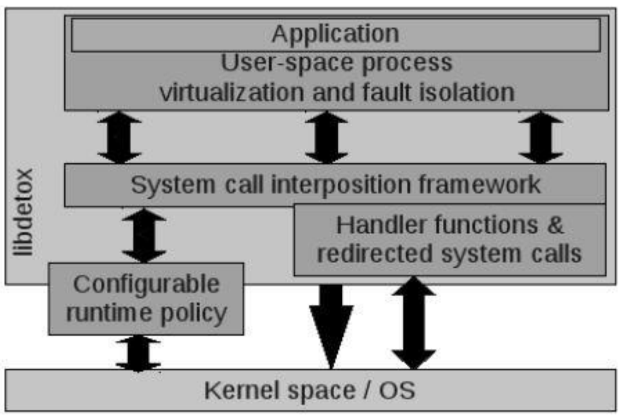


Fig. 7. A process's "Fine-Grained" user-space virtualization layout. (Payer, 2012).

3.5 Supervisor Mode Execution Protection (SMEP)

Supervisor Mode Execution Protection is a recent mitigation technology. SMEP prevents execution out of an untrusted application memory while operating at a more privileged level. SMEP protects against Elevation of Privilege (**EoP**) attacks, and specifically against kernel exploits.
This defensive methods only protects against EoP (Elevation of Privilege attacks), but fails to address Denial of Service and Information Disclosure attacks. Furthermore this defensive method is susceptible to (1) Return-Oriented Programming attacks
(2) Overwriting the "**nt!MmUserProbeAddress**" global variable, which is used for the verification process of memory addresses.

4. Conclusion

Therefore while custom protection techniques attempt to equilibrate the impact of buffer overflow conditions, the most successful approach would be to prevent them before they occur. The accumulation of pointer arithmetic is the reason that languages such as ANSI C and C++ can be subtle to memory related vulnerabilities. Languages such as C can have measurable levels of security if used with an appropriate security life-cycle, but the human factor is accountable, and thus languages such as C, only make it easier for vulnerable code to occur. Thus the hypothesis formed is that high-level languages such as Java to cite as an example, prevent buffer overflows from occurring by design although, the majority of open-source applications and operating-systems have been written in the C and C++ languages, and that is one of the reasons why buffer overflows actually continue to exist. Also approaches such as static and dynamic source code analysis techniques are harder to manage with hundreds of legacy code, and only work to a certain extend requiring manual verification. However, while custom mitigation techniques were developed to reduce the impact from memory exploitation, they do not substitute vulnerable code. Security and robustness both incorporate equal levels of allowable inputs and user driven responses, by limiting user supplied data, so that undesirable outcomes are prevented from circumventing the execution flow. Even if all of the latest technologies are used on a system to the fullest, memory issues will continue to exist.

Although mitigation techniques do considerably reduce the impact, and this is their scope; to reduce the impact to manageable levels. Therefore the future of buffer overflows would gather more interest towards the circumvention of fine-grained user-space virtualization, and perhaps virtualization systems such as Pacifica or HVM. (Levin, 2011). Thus it is submitted that the foreseeable future of memory exploitation techniques will perhaps gather more interest against exploitation of the 64-bit heap, exploitation against novel methods envisaging 'Moving Target Defense' concepts, and such are memory permutation techniques with greater emphasis towards substitution-permutation networks.

References

(Brumley) David Brumley, Mitigating C Vulnerabilities, Carnegie Mellon University, 9 Sep. 2009.

(Cevik) Cevik Serbulent, Memory Corruption Mitigations and Their Implementation Progress in Third-Party Windows Applications, Naval Postgraduate School, Monterey, California Sep 2012.

(Cowan) Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM. Protecting Pointers from buffer overflows.

(Gerardo) Gerardo Richarte.Four different tricks to bypass StackShield and StackGuard Protection, Core Security Technologies, June 2002.

(Negral) Negral, The advanced return-into-libc attacks, 2001.

(VanderVeen) Victor Van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro and Herbert Bos, The Network Institute VU University Amsterdam, Royal Holloway University of London, Memory Errors: The Past, The Present, and the Future, Feb 2012.

(Hiser) Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, Jack W. Davidson, ILR: "Where'd my Gadgets Go?", University of Virginia, Department of Computer Science, 2012.

(Wartell) Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin, Binary Stirring: self-randomizing instruction addresses of legacy x86 Binary Code, Department of Computer Science, The University of Texas at Dallas, October 2012.

(Payer) Mathias Payer, Too much PIE is bad for performance, Department of Computer Science, ETH Zurich, April 2012.

(Payer) Fine-Grained User-Space Security Through Virtualization, ETH Zurich, 2011.

(Limin) Limin Liu, Jin Han, Debin Gao, Jiwu Jing, Daren Zha, Launching Return-Orientated Programming attacks against Randomized Relocatable Executable, State Key Laboratory of Information Security Graduate University of CAS Beijing, China, 2012.

(Schwartz) Edward J. Schwartz, Thanassis Avgerinos and David Brumley, Exploit Hardening Made Easy, Carnegie Mellon University, Pittsburgh, PA, 2012.

(Piromsopa) Kerk Piromsopa, Richard J.Enbody, Survey of protections from buffer overflow attacks, Department of Computer Science and Engineering, Michigan State University, Engineering Journal, Issue 2, Vol 15, April 2011.

(Fuchaco) Yan Fen,Yuan Fuchaco, Shen Xiaobing, Yin Xinchun,Mao Bing A New Data Randomization Method to Defend Buffer Overflow Attacks, 2012 International Conference on Applied Physics and Industrial , Monterey, California Sep 2012.

(Tian Shuo), He Yeping, Ding Baozeng, Prevent Kernel Return-Orientated Programming Attacks Using Hardware Virtualisation, Institution of Software, Chinese Academy of Sciences, 2012.

(Shacham) Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modagugu, Den Boneh On the Effectiveness of Address-Space Randomization, Stanford University, 2004.
Usenix Security Symposium, 2003

(Monica) Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization.
Technical report, UC Berkeley, 2002.

(Sotirov) Alexander Sotirov, Bypassing Memory Protections: The Future of Exploitation.

(Brumley) D.Brumley, The Security of Address Space Layout Randomization (ASLR), Carnegie Mellon University, 2010.

(Levin) Warren Levin, Exploitation in a hostile world, 2011.

(Jurczyk) Mateusz Jurczyk,
SMEP: What is it and how to beat it on Windows, 2011.

(Yutaka) Oiwa Yutaka, Fail-Safe ANSI-C Compiler: An Approach to Making C Program Secure, University of Tokyo, Japan Science and Technology Corporation.

(Zheng Xu) Zheng-Xu Zhao David J Day Protecting Against Address Space Layout Randomisation (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems, University of Derby, 2011.