

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang

Presented by Jacky Mok

Agenda

- Background
 - Buffer overflow
 - Stack smashing
- Stackguard
 - Goal
 - Implementation
 - Results and conclusion
- Pros and cons
- Discussion

Buffer Overflow

- Invalid input to a function that causes the program to overwrite memory that is not supposed to be re-written, causing machine to behave abnormally
- Cause of threat
 - Unsafe-ness of programming language (C)
 - Programming error
 - Insufficient boundary check
- Most common form is stack smashing

Stack

Largely simplified stack
For a basic block

Stack

Mem	Code
0x0	function f
0x4	get a from stdin
0x8	x = a
0xC	return x
0x10	end
0x14	Main
0x18	call f
0x1C	end

Stack for f

For simplicity, assume each instruction and data type takes 4 bytes

Stack smashing

- Say, I input 'a' to be a char array of "4 4 0x20"
 - Variable a now takes 12 bytes instead of 4

Static stack allocation

Run time value

Stack smashing

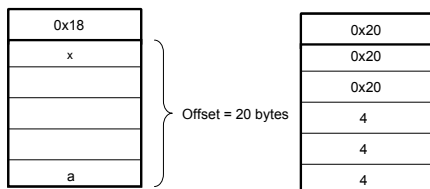
- The function `f` now returns the control to whatever is sitting on `0xF0`, it could cause the program to:
 - Core dump (memory is out of scope)
 - Invalid results (memory contains garbage)
 - Lose control (memory points to an instruction that returns control to hacker, i.e. open a shell, spread virus, etc...)

Stack smashing

- 2 mutually dependent parts
 - Changing the return address
 - Installing the code

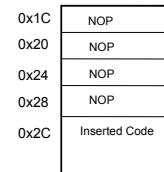
Is it hard to change the ret addr?

- It comes out that hackers don't need to know the exact offset of the return address
- They could repeat the same garbage value for `a` to figure out the offset



Is it hard to know the start addr of the inserted code?

- It comes out that hackers don't need to know the exact starting addr of the inserted code
- They could insert some NOPs before the start, and the starting addr just need to hit onto one of the NOPs



Stack guard

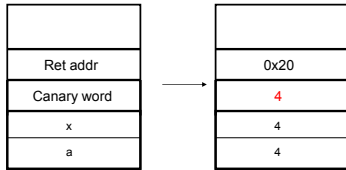
- A compiler extension that inserts special instructions into each basic block to guard stack smashing
- Halts program when stack smashing is detected
- Needs to recompile source code

Stack guard

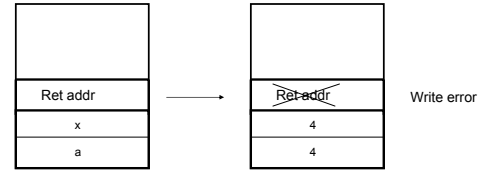
- Facilitates 2 defensive mechanisms, canary word and memGuard

Canary: detects attack, minimal performance hit
memGuard: prevents attack, degrades performance

Canary



MemGuard



Experimental results

Vulnerable Program	Result w/o StackGuard	Result with Canary StackGuard	Result with MemGuard StackGuard
dip	root shell	halts	halts
elm	root shell	halts	halts
Perl	root shell	halts irregularly	root shell
Samba	root shell	halts	halts
SuperProbe	root shell	halts irregularly	halts
umount / libc	root shell	halts	halts
wwwcount	htp shell	halts	halts
zgv	root shell	halts	halts

Performance on micro benchmark

Increment method	Standard Runtime	Canary	MemGuard register	MemGuard VM
i++	15.1	15.1	15.1	N/A
void inc()	35.1	60.2	1808	34900
void inc (int *)	47.7	70.2	1820	40420
int inc (int)	40.1	60.2	1815	41610

Performance on macro benchmark

Input	Version	User time	System time	Real time
37,000 lines into ctags	Generic	0.41	0.14	0.55
	Canary	0.68	0.13	0.99
	MemGuard Register	1.30	5.45	6.84
	MemGuard VM	16.5	238.0	255.1
586,000 lines into ctags	Generic	7.74	2.08	10.2
	Canary	11.9	2.07	14.5
	MemGuard Register	21.1	91.5	115
	MemGuard VM	236	3482	3728
GCC	Generic	1.70	0.12	1.83
	Canary	1.79	0.16	1.96
	MemGuard Register	2.22	3.35	5.76
	MemGuard VM	8.17	87.7	96.2

Pros

- Robust and elegant solution
- Should work well with other programs
- Efficient and effective

Cons

- Feasibility:
 - Not a feasible solution to address legacy code to avoid buffer overflow attacks
 - Legacy programs might not have the source code to be recompiled
 - Cost to recompile an application is high:
 - Re-testing
 - Planning
 - Re-certificate
 - Redistribute
 - The Canary version is potentially subject to guessing of the canary value. Alternatives to make it harder for the attacker to guess is to enhance the crt0 library to choose a set of random canary words at the time the program starts.
 - Users needs to use the enhanced crt0 library
 - Runtime incompatibility issues

Cons

- Performance
 - The MemGuard variant of StackGuard also suffered a large performance degradation. The use of MemGuard can introduce as large as 174,300% overhead in a regular function call.
 - Maximum benefit that can be provided by StackGuard is impractical even for debugging purposes.
- Does not stops all overrun attacks
 - Not successful on non-sequential overwrites
 - Protects only return addresses
 - Uses a buffer overflow to overwrite a function pointer that is on the heap

Cons

- Overall, compiled StackGuard application cannot be use to help programs that are already in use in the market
- Canary variant can be used for new programs but MemGuard variant is too costly to use due to its runtime overhead

Summary

Q n A

Dynamic Taint Analysis for
Automatic Detection, Analysis, and
Signature Generation of Exploits on
Commodity Software

Presented by Rita Chiu

Objective

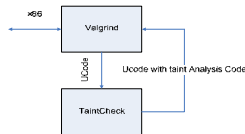
- To combat fast Internet worm attacks by providing an automatic detection mechanism that can detect new attacks from previously unknown vulnerabilities.
 - Easy to deploy.
 - Few false negatives / positives.
 - Automatically develop filters (a.k.a. attack signatures) that can be used to filter out attack packets efficiently, and protect vulnerable hosts from compromise until a patch is available.

Overview

- TaintCheck performs dynamic taint analysis on a program by performing binary rewriting at runtime that work on commodity software
 - No source code / recompilation is required (easy to deploy)
- TaintCheck runs the program in its own emulation environment
 - Allows to monitor and control the program's execution at a fine-grained level.
 - Less false positives
 - More detailed information about the vulnerability and exploit.

Overview

- The emulator: Valgrind
- An open source x86 emulator
- Supports extensions, called skins
- Able to instrument a program as it runs



Overview

- Legitimate assumption:
 - To change the execution of a program illegitimately, a value that is normally derived from a trusted source is now derived from an attacker's own input.
 - For example: jump addresses and format strings should usually be supplied by the code itself, not from external untrusted inputs.
 - Any data originated from or arithmetically derived from untrusted sources (e.g. the network) is labeled as tainted.

Proposed Techniques

- TaintCheck is broken down into 4 components
 - TaintSeed
 - Mark data as untrusted source as tainted
 - TaintTracker
 - Monitors how the tainted data propagates during the program's execution time.
 - TaintAssert
 - Detection of attacks
 - Exploit Analyzer
 - Derived information on the newly detected attacks

Proposed Techniques (TaintSeed)

- TaintSeed
 - Marks data from untrusted source as tainted
 - Examines the arguments and results of each system call, and determines whether any memory written by the system call should be marked as tainted or untainted according to the TaintSeed policy.
 - By default, any data from the network is tainted
 - Each byte of memory has a 4-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not.
 - Taint data structure includes information such as:
 - The system call number
 - A snapshot of the current stack
 - A copy of the data that was written is recorded when the memory is tainted

Proposed Techniques (TaintTracker)

- TaintTracker
 - Monitors program execution to track how the tainted attribute propagates and determine whether the result is tainted.
 - Tainted result is sets to have its shadow memory to point to the same Taint structure as the tainted operand.
 - Allows the Exploit Analyzer to follow this chain of Taint structures backwards to determine how the tainted data propagated through memory.

Proposed Techniques (TaintAssert)

- TaintAssert
 - Indicates when tainted data is used in “dangerous ways”.
 - Default policy detects:
 - Jump addresses
 - Format
 - Optional:
 - System call arguments
 - Application or library-specific checks

Proposed Techniques (TaintAssert)

- Jump addresses (default):
 - Overwrite the address in an attempt to redirect control flow to the attacker’s code, or another point in the program
 - TaintCheck place instrumentation before each UCode jump instruction to ensure that data specifying the jump target is not tainted.
- Format strings (default):
 - Trick a program into leaking data or into writing an attacker-chosen value to an attacker-chosen memory address by providing a malicious format string.
 - Wrappers are used to wrap around the printf family of functions to ensure that the format string is not tainted (and optionally, does not contain format specifier %n), and then call the original function.
 - This will not catch any format strings attack if the program uses its own implementation of these functions because the wrappers may not be called.

Proposed Techniques (Exploit Analyzer)

- Exploit Analyzer
 - Automatically provides information about the vulnerability, how the vulnerability was exploited, and which part of the payload led to the exploit of the vulnerability.
 - Uses information from TaintSeed and TaintTracker to derive information such as:
 - Original input buffer that tainted data came from
 - Program counter
 - Call stack at every point the program operated on the relevant tainted data
 - The point where the exploit occurred.
 - Attack signatures (3 most significant bytes of a value used to overwrite a jump target)

Results

- No false positives for any of the many different programs that have tested under the default policy
- The number of false Positive depends on how the tainted data policy is configured.

Tainted Data	Default/Optional	False Positive?	Workaround?
Network	Default	No	--
Standard Input	Optional	Yes	Reconfigure to trust the cfg files.
Files	Optional	Yes	Reconfigure to trust the cfg files.

Results

- Detects exploits correctly

Program	Overwrite Method	Overwrite Target	Detected
ATPhhttpd	Buffer overflow	Return address	✓
synthetic	Buffer overflow	Function pointer	✓
synthetic	Buffer overflow	Format String	✓
synthetic	Format string	None (info leak)	✓
cfingerd	syslog format string	GOT entry	✓
wu-ftpd	vsprintf format string	Return address	✓

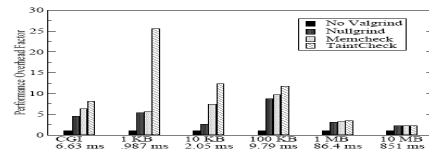
Results

- Performance evaluation:
 - Used the worst-case workloads (i.e. CPU-bound and short-lived process workload)
 - CPU intensive
 - Large initialization cost

	Normal (sec)	Valgrind's Nullgrind skin (sec)	MemCheck (sec)	TaintCheck (sec)
CPU-bound (bzip2)	8.2	25.6 (3.1 times)	109 (13.3 times)	305 (37.2 times)
Short-lived (cfingert)	0.0222	0.2886 (13.0 times)	0.7104 (32.0 times)	0.7992 (36.0 times)

Results

- Performance evaluation:
 - Common case: Apache (Long-lived and I/O bound), therefore, does not show as big of an overhead as the



Improvements

- Use a more optimized emulator than Valgrind.
 - 3.1 times slower when a Nullgrind is used
- Statically analyze each basic block to eliminate redundant tracking code.
 - Reduces the amount of instrumentation added

Deployment

- TaintCheck cannot be used all the time
 - Performance overhead
- Use in conjunction with a faster detector
 - Reduces its false positive rate
 - Provides additional information of an attack
 - Examples:
 - TaintCheck-enabled honeypots
 - TaintCheck plus OS randomization

Future Work

- Investigate more advanced techniques of semantic analysis to assist automatic signature generation
 - Keep track of whether each byte of the request is used in any significant way, and how it is used.
 - An additional performance overhead

Pros

- Contributions:
 - Automatically detect, analysis, and signature generation of exploits on commodity software.
 - Does not require source code or specially compiled binaries. This is more convenient to be deployed on commodity software.
 - Generates signatures automatically and requires fewer attack payloads to generate signature. This enables the community to be aware of any new attack faster.
 - TaintCheck can be customized according to the users' need.
- Performance overhead:
 - Stated challenges clearly. Provided alternatives to enhance the performance is also discussed in the paper.
- Provides promising results with few false negatives and positive despite the performance overhead.
- Feasible if higher processing power is used in the commodity hardware.

Cons

- Insufficient testing
 - Very good coverage on no-false positive
 - but this could mean the program doesn't do anything
 - Moderate coverage on correctness,
 - 1 macro benchmark with 1 input
 - 2 micro benchmark with limited input
 - 3 synthetic attacks
 - Moderate coverage on performance,
 - Limited results
- Bad performance
 - Overhead factor, min 2.5, max 26
 - Emulator's fault?
- Inadequate proofs on framework
 - Does an attacker always require to change the memory to facilitate an attack?
 - Is a change of memory always an attack?

Q n A

N-Variant Systems
A Secretless Framework for Security through Diversity
Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser
Presented by Jacky Mok

Agenda

- Background
- Motivations
- Model
- Examples
- Performance
- Summary

Background

- Attackers take advantage of the homogeneity of systems
- One defensive mechanism is to make each system different to the attackers, making them harder to infect and limit the propagation
- Previous attempts are to store a private key – but this has been proven to be vulnerable
- Attacks propagates quickly, need a defense that can guard attack efficiently

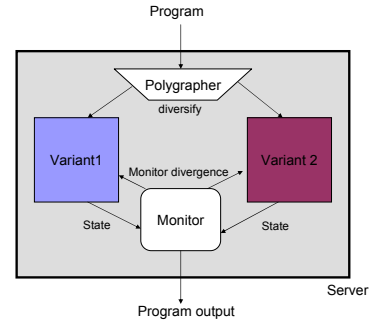
Motivations

- There are a lot of vulnerability classes
- Goal is to develop a scalable system to defend against all classes of attacks (potentially)
- New system makes N copies of the program, each with an identifiable feature.
- For an attack to fly under the radar, it has to compromise all variants together

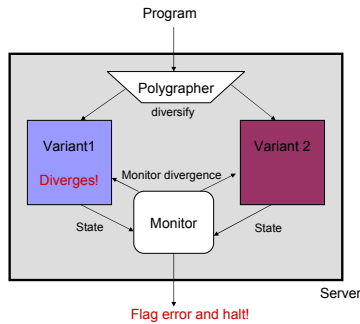
N-variant systems

- Creates N copies (variants) of the program, each variant on a different exploitation set and observe the behavior.
- Under normal circumstances and good programming practice, each variant should behave the same way
- Under an attack, one or more variants should diverge from the others

Model



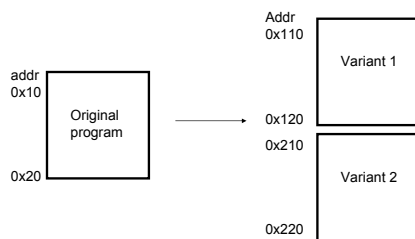
Model



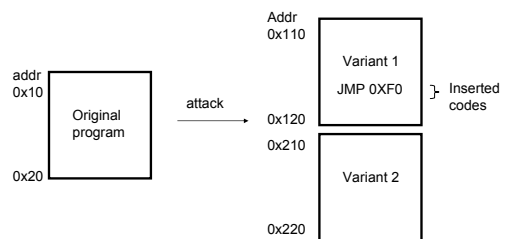
Premises

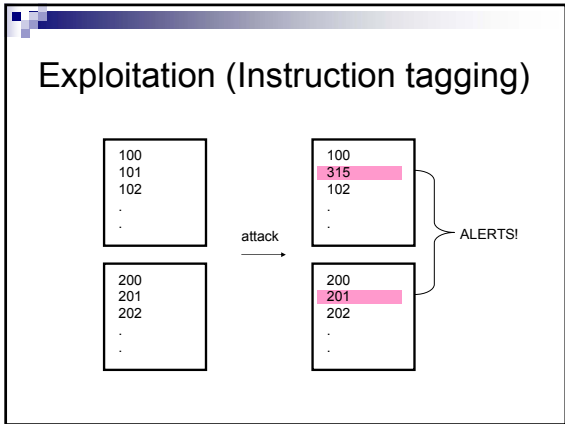
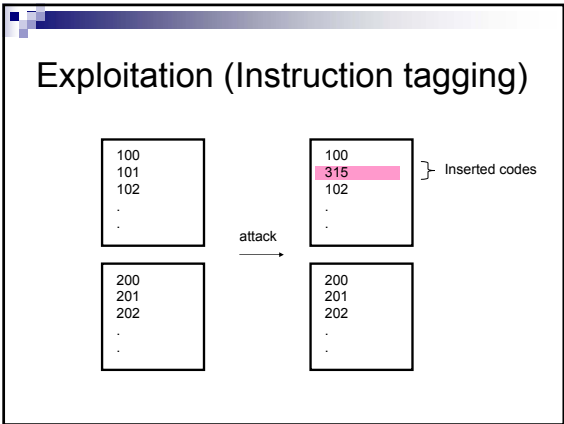
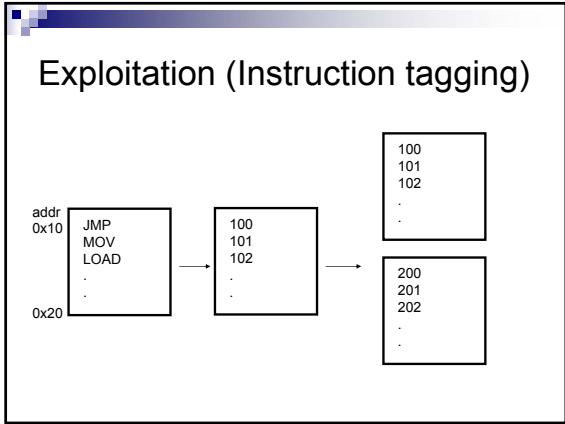
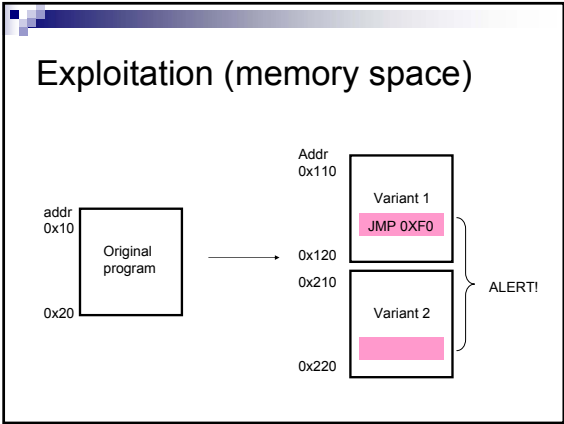
- Normal Equivalence
 - Inverting variants will yield the same state in the original program
- Detection
 - At least one variant must raise alerts if one or more variants are compromised

Exploitation (memory space)



Exploitation (memory space)





Test results

Configuration		1	2	3	4	5	6
Description		unmodified	Unmodified apache, n-variant kernel	2-variant system, address partitioning	Apache running strata	Apache with instruction tags	2-variant system, instruction tags
Unsaturated	Throughput (MB/s)	2.36	2.32	2.04	2.27	2.25	1.80
	Latency (ms)	2.35	2.40	2.77	2.42	2.46	3.02
saturated	Throughput (MB/s)	9.70	9.59	5.06	854	8.30	3.55
	Latency (ms)	17.65	17.80	34.20	23.30	20.58	48.30

- ### Pros
- Defend against attack classes rather than specific exploits in programs
 - Solution is portable between programs
 - Should work with other vulnerability detection programs

Cons

- Cannot describe how to detect multiple types of attacks
 - Detection depends on the type of variation used
 - Each variation can detect only a certain class of known attacks
 - Implementation overhead
 - Need to ensure both normal equivalence and detection properties are satisfied
 - Might not be able to quickly prevent new type of attacks

Cons

- Proposed variations to detect attacks are not complete
 - Address Space Partitioning
 - Does not account for attacks against memory corruption that depends on relative address
 - Many existing randomization solutions provides a more complete memory corruption detection than this secret-less approach
 - E.g. Address Obfuscation

Cons

- False positives rate not provided
 - False positives could occur when the monitor observes differences between the outputs for normal requests
 - Output web pages includes a time stamp or an IP address can introduce false positives
- Requires changes to OS kernel

Cons

- Performance
 - Performance degrades proportion to the number of variants in the system
 - Throughput degrades by 50% in 2-variant system because computational cost is doubled
 - Performance depends on the variation used
 - Only 2 variations are described

Summary...

Q n A

Summary

Summary

- Detects at least the following attacks

	Buffer Overflow		Format String	Function Pointers	Double Free
	Injected code	Change of return address			
StackGuard	√	√			
Dynamic Taint Analysis	√	√	√	√	√
N-Variant	√ (if instruction set tagging is used)	√ (on changed in absolute address only)	√ (on changed in absolute address only)	√ (on changed in absolute address only)	√ (on changed in absolute address only)

Summary

- Deployment overhead

	Change of compiler	Change of OS	Use of emulator
StackGuard	√	√ (crt0)	
Dynamic Taint Analysis			√
N-Variant		√	

Summary

- Vulnerabilities, implementation/performance overhead, enhancement

	Can this be defeated by an attacker?	Implementation/Performance overhead?	Enhancement?
StackGuard (Canary variant)	YES. By guessing the canary value.	Low	Randomizing the canary value.
Dynamic Tainted	YES, but hard and not systematic.	High (Performance)	Use a more efficient emulator.
N-Variant	YES. Use relative address.	High (Implementation)	No.