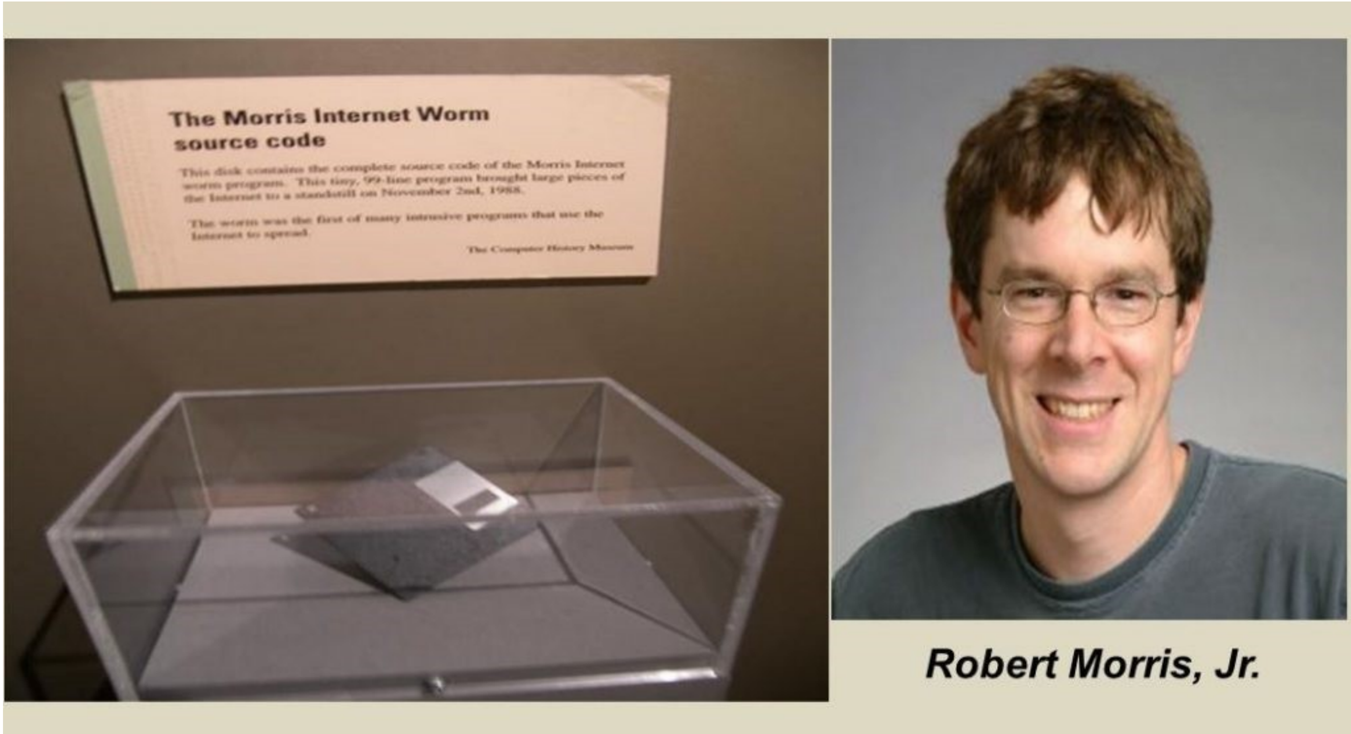
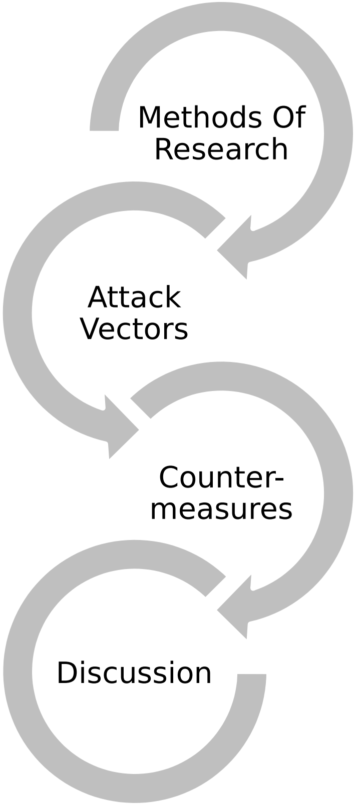

The Morris Internet Worm
source code
This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.
The worm was the first of many intrusive programs that use the Internet to spread.
The Computer History Museum
Robert Morris, Jr.

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

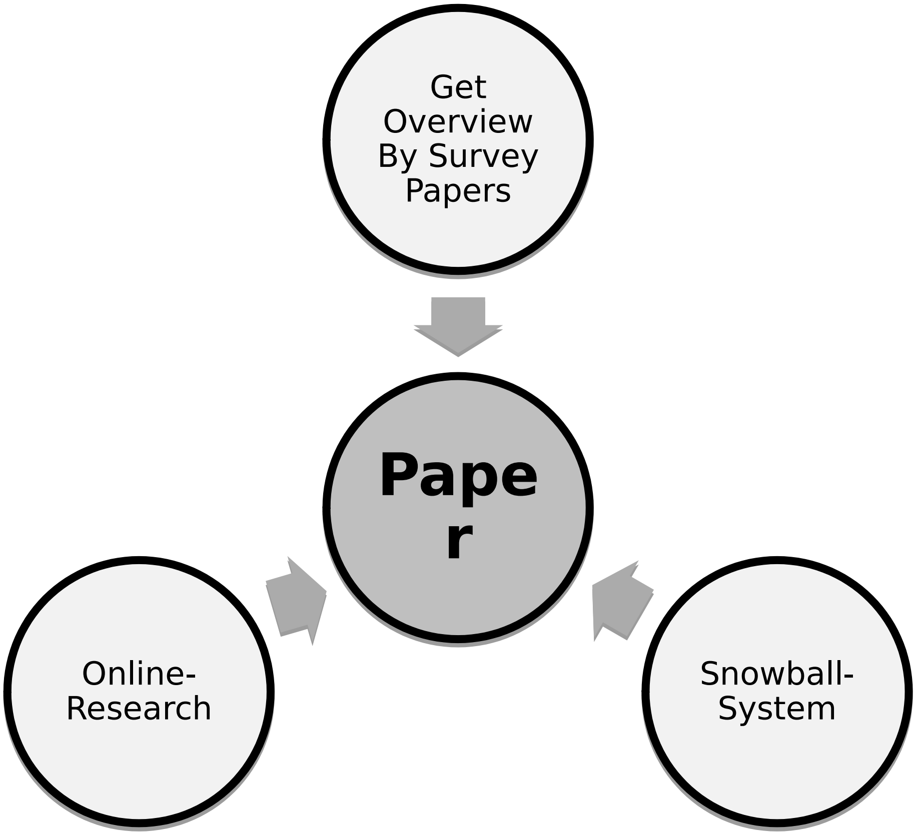
Overview Over Attack Vectors And Countermeasures For Buffer Overflows

Christian Müller | Julian Dietrich | Valentin Brandl
Faculty of Computer Science and Mathematics
OTH Regensburg

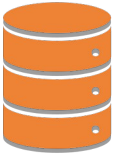
Agenda


Methods Of Research
Attack Vectors
Counter-measures
Discussion

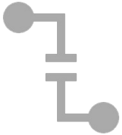
Methods Of Research



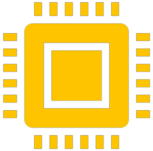
Attack Vectors



Stack-based buffer overflows



Heap-based buffer overflows

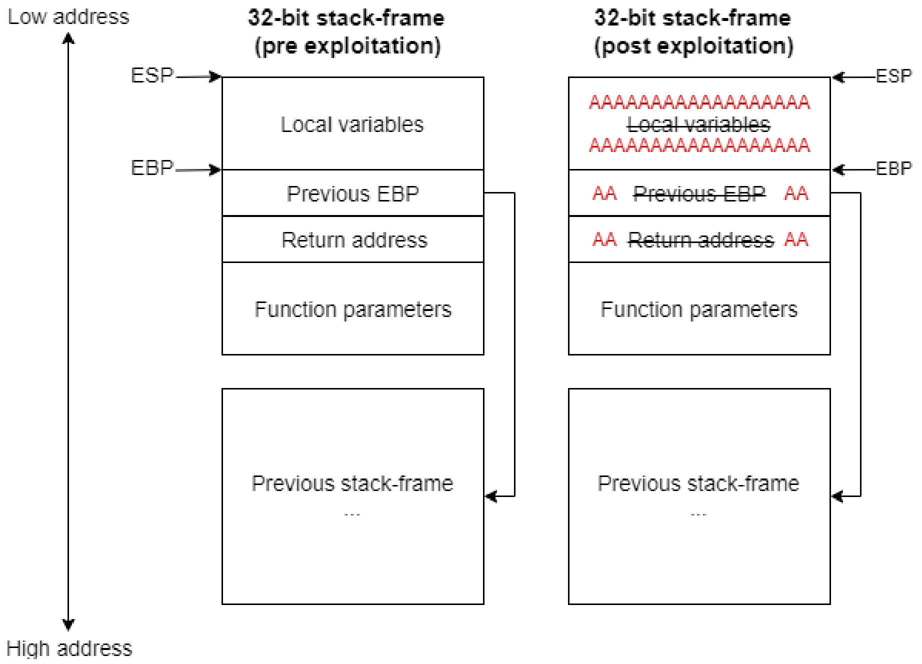


Integer overflows

Stack-based buffer overflows

- Stack contains: Parameters, Local Variables, Return Address, …
- Return Address: Next address to execute when called function returns
- Local Variables: Can contain function pointers
- **General Goal:** Overwriting *Return Address* or *Local Function Pointers* to gain <u>Code Execution</u>

Stack-based buffer overflows



Heap-based buffer overflows

- Heap contains: Class Instances, Function Pointers, Heap Metadata, …
- Heap Metadata: Used by Heap Management operations such as *freeing*, *merging*, *splitting* chunks
- Type confusion: Modify internal Object Type stored by dynamic typing languages such as Python or JavaScript
- **General Goal:** Overwriting *Function Pointers* or *Heap Metadata* to gain Code Execution

Integer overflows

- Does not directly lead to Code Execution

- Used to trigger Heap-based BOFs (buffer overflows)
  → Overflow integer which determines allocation size
  → Integer is smaller than needed size
  ➔ Out of bounds access

- **General Goal:** Triggering a Heap-based BOF to gain Code Execution

ASLR

- Randomize location of program in memory
- Attacker doesn‘t know where payload is located
- Prevents code execution
- Information leak allows exploitation
- Brute-force of 32 bit addresses possible
- Does not prevent DoS
- Compile-time mitigation, no code changes needed

ASLR

```c
#include<stdio.h>

void some_function() {
  puts("Hello, world!\n");
}

int main() {
  void (*function)() = &some_function;
  printf("some_function is located at %p\n", function);
  return 0;
}
```

```
~/wis
➜ gcc example.c -o example

~/wis
➜ ./example
some_function is located at 0x55a96720b149

~/wis
➜ ./example
some_function is located at 0x555f0c1ff149
```

```
~/wis
➜ gcc example.c -o example -no-pie

~/wis
➜ ./example
some_function is located at 0x401136

~/wis
➜ ./example
some_function is located at 0x401136
```

NX

- Memory can be either writable or executable
- Attacker cannot supply shellcode directly
- Code reuse still possible
- Compile-time mitigation, no code changes needed

Stack Canary

- Markers at the end of a stack frame
- Invalid marker → Buffer overflow occurred
- No code changes required
- Only mitigates stack-based BOF
- Knowledge of canary allows bypassing

RAD

- Read-only stack for return addresses
- Compared before return
- Compiler extension
- Only against stack-based BOF

- Each indexing operation is checked
- 100% effective (where applied)
- Non-trivial runtime overhead
- Used in languages with runtimes (Java, C#, Python, ...)

Dependent Types

- Value (size) is associated with a buffer
- Only allow indexing with validated values
- Language extension
- Lot of work to use, but type inference helps

Discussion – Static Methods

+

Wrong Positives

-

Know-How

Costs

Access To Source-Code

Discussion – Static Methods

+

- Wrong Positives

-

- Know-How
- Costs
- Access To Source-Code

Discussion – Static Methods

+

- Wrong Positives

-

- Know-How
- Costs
- Access To Source-Code

Discussion – Static Methods

+

- Wrong Positives

-

- Know-How
- Costs
- Access To Source-Code

Discussion – Static Methods

+

- Wrong Positives

-

- Know-How
- Costs
- Access To Source-Code

Discussion – Dynamic Methods

+

- Implement-ation

-

- Wrong Positives
- Access To Source Code
- Legacy Code
- Size Of Memory

Discussion – Dynamic Methods

+

- Implement-ation

-

- Wrong Positives
- Access To Source Code
- Legacy Code
- Size Of Memory

Discussion – Dynamic Methods

+

- Implement-ation

-

- Wrong Positives
- Access To Source Code
- Legacy Code
- Size Of Memory

Ref.: [1], [8], [9], [10], [11]

Discussion – Dynamic Methods

+

- Implement-ation

-

- Wrong Positives
- Access To Source Code
- Legacy Code
- Size Of Memory

Ref.: [1], [8], [9], [10], [11]

Discussion - Conclusion

- Until today, a lot of software is developed in unprotected languages

- Combination if techniques provides best results
  - Computational intelligence combined with static methods

Forecast

- More computational intelligence techniques
- Techniques to handle buffer overflow vulnerabilities automatically

Ref.: [1], [8], [9], [10], [11]

References (1/2)

[1] M. L. Chaim, D. S. Santos, and D. S. Cruzes, “What do we know about buffer overflow detection? a survey on techniques to detect a persistent vulnerability,” International Journal of Systems and Software Security and Protection (IJSSSP), 2018.

[2] W. Wang, “Survey of attacks and defenses on stack-based buffer overflow vulnerability,” Advances in Computer Science Research (ACSR), 2017.

[3] Y. Younan, W. Joosen, and F. Piessen, “Runtime countermeasures for code injection attacks against c and c++ programs,” ACM Computing Surveys (CSUR), 2012.

[4] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song, “Rich: Automatically protecting against integer-based vulnerabilities (rich),” RICH Journal Group (RICH), 2007.

[5] W. Dietz, P. Li, J. Regehr, and V. Adve, “Understanding integer overflow in c/c++,” ACM Trans. Softw. Eng. Methodol. (TOSEM), vol. 25, no. 1, pp. 2:1–2:29, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2743019

[6] D. Binkley, “Source code analysis: A road map,” IEEE Computer Society (IEEE CS), 2007.

References (2/2)

[7] M. Harman, “Why source code analysis and manipulation will always be important,” IEEE Computer Society (IEEE CS), 2010.

[8] B. M. Padmanabhuni and H. B. K. Tan, “Buffer overflow vulnerability prediction form x86 executables using static analysis and machine learning,” in 2015 IEEE 39th Annual International Computers, Software and Applications Conference (IEEE), 2015.

[9] M. Dalton, H. Kannan, and C. Kozyrakis, “Real-world buffer overflow protection for userspace and kernelspace,” in USENIX Security Symposium (USENIX), 2008.

[10] O. Ruwase and M. S. Lam, “A practical dynamic buffer overflow detector,” in The Network and Distributed System Security Symposium (NDSS), 2004.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in USENIX Security Symposium, 1998.

[12] https://computersciencementor.com/difference-between-virus-and-worm/, entnommen: 14.12.19.