Graph Neural Network based Cache Monitor for ROP Attack Detection

Asmit De and Saptarashmi Bandyopadhyay

School of Electrical Engineering and Computer Science
The Pennsylvania State University

Outline

- Introduction
  - Buffer Overflow Vulnerability
  - Existing Defense Mechanisms
  - Return Oriented Programming
- Cache Monitoring for ROP Attacks
  - ROP Chain Creation
  - Cache Statistics
  - Implementation
  - Case Study: Nginx
- Machine Learning Based Detection Technique
  - Motivation
  - Support Vector Machine
  - Random Forest
  - Graph Neural Networks
  - Training
  - Testing
  - Results
- Conclusions

Introduction

- Buffer Overflow Vulnerability
- Existing Defense Mechanisms
- Return Oriented Programming

Buffer Overflow Vulnerability

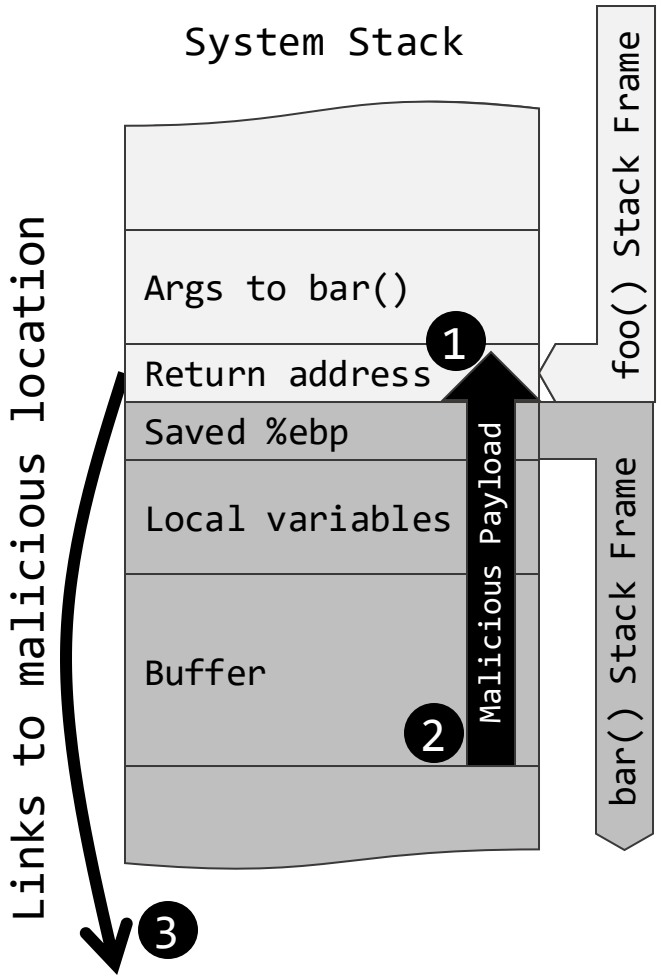
- Buffer overflow is common in low level languages such as C
- Primary Issue: Lack of bounds checking
- `memcpy` writes `len` bytes to the buffer, but `len` can be adversary controlled.
- `len > 32` overwrites data above the buffer: such as variables `i`, and even saved base pointers and return addresses.
- Exploits: Return-Oriented Programming (ROP), Function-pointer manipulation, CFI and DFI violations violations.

```
int vuln_fn( char *data, int len )
{
   int i = 0;
   char buf[32];

   memcpy( buf, data, len );

   return 0;
}
```

Defense Mechanisms

- Stack Canaries
  - Sacrificial words placed at stack return boundaries
  - If adversary overwrites the return address, canary is also modified, and thus detected
  - Issues: Disclosure vulnerability can leak the canary. Adversary can jump over the canary using a function pointer
- Data Execution Prevention (DEP)
  - Mark memory pages as $W \oplus X$, so they can be either executable (code), or writable (stack, heap), but not both
  - Prevents adversary form running maliciously injected code from the stack
  - Issues: Adversary can still return to existing code in the program or libc library functions
- Address Space Layout Randomization (ASLR)
  - Randomize the base addresses for the stack, heap, code, libraries, etc.
  - Issues: Adversary can still find out the offsets through a memory disclosure or side-channels
- Secure Hardware Platforms
  - Intel SGX, CET, TSX, MPX, ARM TrustZone etc. can be used to protect memory
  - Issues: Applications need to specifically use the hardware protection. Not scalable.

Return Oriented Programming



foo():

- *Some code*
- Push args for bar()
- Push return address on stack
- Jump to bar()

1

bar():

- *Some code (adversary injects payload here)*
- Begin bar() epilogue actions
- Jump to malicious location

2

3

Malicious location can be:

- Adversary injected code in stack
- Existing code from function epilogs (ROP Gadgets)
- Code from libc library, etc.

Cache Monitoring for ROP Attacks

- ROP Chain Creation
- ROP Analysis
- Cache Statistics
- Case Study: Nginx

ROP Chain Creation

```c
#include <stdio.h>

void overflow(char *inbuf, int len)
{
   char buf[4];
   memcpy(buf, inbuf, len);
}

int main(int argc, char **argv)
{
   overflow(argv[1], strlen(argv[1]));
   return 0;
}
```

```
ROP chain generation
===========================================================

- Step 1 -- Write-what-where gadgets

   [+] Gadget found: 0x47eea1 mov qword ptr [rsi], rax ; ret
   [+] Gadget found: 0x4100c3 pop rsi ; ret
   [+] Gadget found: 0x4150c4 pop rax ; ret
   [+] Gadget found: 0x444620 xor rax, rax ; ret

- Step 2 -- Init syscall number gadgets

   [+] Gadget found: 0x444620 xor rax, rax ; ret
   [+] Gadget found: 0x4742f0 add rax, 1 ; ret
   [+] Gadget found: 0x4742f1 add eax, 1 ; ret

- Step 3 -- Init syscall arguments gadgets

   [+] Gadget found: 0x400686 pop rdi ; ret
   [+] Gadget found: 0x4100c3 pop rsi ; ret
   [+] Gadget found: 0x4492e5 pop rdx ; ret

- Step 4 -- Syscall gadget

   [+] Gadget found: 0x40122c syscall

- Step 5 -- Build the ROP chain
```

ROP Analysis

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack
# Padding goes here
p = ''
p += "A"*16
p += pack('<Q', 0x00000000004100c3)  # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e0)  # @ .data
p += pack('<Q', 0x00000000004150c4)  # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x000000000047eea1)  # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x00000000004100c3)  # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e8)  # @ .data + 8
p += pack('<Q', 0x0000000000444620)  # xor rax, rax ; ret
p += pack('<Q', 0x000000000047eea1)  # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x0000000000400686)  # pop rdi ; ret
p += pack('<Q', 0x00000000006b90e0)  # @ .data
p += pack('<Q', 0x00000000004100c3)  # pop rsi ; ret
p += pack('<Q', 0x00000000006b90e8)  # @ .data + 8
p += pack('<Q', 0x00000000004492e5)  # pop rdx ; ret
p += pack('<Q', 0x00000000006b90e8)  # @ .data + 8
p += pack('<Q', 0x0000000000444620)  # xor rax, rax ; ret
p += pack('<Q', 0x0000000004742f0)  # add rax, 1 ; ret
p += pack('<Q', 0x0000000004742f0)  # add rax, 1 ; ret
...
p += pack('<Q', 0x0000000004742f0)  # add rax, 1 ; ret
p += pack('<Q', 0x0000000004742f0)  # add rax, 1 ; ret
p += pack('<Q', 0x000000000040122c)  # syscall
print p
```

59 same inst

Syscall #59 is sys_execve!

%rdi : pointer to filepath

%rsi : arguments

%rdx: environment variables

20,481 bytes or ~20KB

433,629 bytes or ~423KB

214,365 bytes or ~209KB

239,745 bytes or ~234KB

195,792 bytes or ~191KB

471,236 bytes or ~460KB

Cache Statistics

- We ran Valgrind-Cachegrind tool to collect the following cache signatures as features by running nginx
  - I references, D references, LL references
  - I1 misses, LLi misses, D1 misses, LLd misses, LL misses
  - I1 miss rate, LLi miss rate, D1 miss rate, LLd miss rate, LL miss rate

- We are collecting 13 features for each of the 5600 data executions comprising of cache references, cache misses and cache miss rate

- Trace collection
  - Injected a ROP vulnerable function at different positions of the code
  - Collected cache statistics for 100 runs of normal execution for each position
  - Collected cache statistics for 100 runs of execution for each position when under a ROP attack

Case Study: Nginx

Normal Execution

```
I   refs:      1,208,494
I1  misses:        3,694
LLi misses:        3,060
I1  miss rate:      0.31%
LLi miss rate:      0.25%

D   refs:        412,618  (287,672 rd   + 124,946 wr)
D1  misses:        6,775  (  3,973 rd   +   2,802 wr)
LLd misses:        4,738  (  2,232 rd   +   2,506 wr)
D1  miss rate:       1.6% (    1.4%     +     2.2%  )
LLd miss rate:       1.1% (    0.8%     +     2.0%  )

LL refs:          10,469  (  7,667 rd   +   2,802 wr)
LL misses:         7,798  (  5,292 rd   +   2,506 wr)
LL miss rate:        0.5% (    0.4%     +     2.0%  )
```

Execution under ROP Attack

```
I   refs:      1,211,567
I1  misses:        3,748
LLi misses:        3,063
I1  miss rate:      0.31%
LLi miss rate:      0.25%

D   refs:        413,559  (288,155 rd   + 125,404 wr)
D1  misses:        6,782  (  3,973 rd   +   2,809 wr)
LLd misses:        4,742  (  2,233 rd   +   2,509 wr)
D1  miss rate:       1.6% (    1.4%     +     2.2%  )
LLd miss rate:       1.1% (    0.8%     +     2.0%  )

LL refs:          10,530  (  7,721 rd   +   2,809 wr)
LL misses:         7,805  (  5,296 rd   +   2,509 wr)
LL miss rate:        0.5% (    0.4%     +     2.0%  )
```

PennState
College of Engineering

Machine Learning based Detection Technique

- Motivation
- Support Vector Machine
- Random Forest
- Graph Neural Networks
- Training
- Testing
- Results

Motivation for Machine Learning

- Hypothesis: Return-oriented Programming generates a distinctive cache signature that is different from a normal execution of the same binary
- Our goal has been to identify the features necessary for classification of the binary execution as normal or vulnerable
- We have implemented a linear Support Vector Machine classifier and an ensemble based classifier (Random Forest) to understand the role of features in separating these data points
- We have applied the concept of Graph Neural Networks (GNN) for the 1st time in the Hardware Security domain
- Our intuition of GNN is to understand the dependence of cache signatures when ROP based payload is introduced in different blocks of the source code to improve the classification

Training and Testing Dataset

- There are 5600 data points for each of the 28 positions (pos<i>) where RoP based payload has been introduced.
- For each (pos<i>) there are 100 data points for normal execution and 100 for vulnerable execution, which are generated by repeated execution of the cachegrind tool
- We have implemented a 80/20 split in training and testing data for SVM and Random Forest classifier
- For GNN, we have done a 70/10/20 split in training, validation and testing dataset
- We have created a graph where each of the 5600 data points are nodes. Each of the 200 nodes for pos<i> are connected with the 5400 data points in the remaining 27 positions
- This has been done to study the impact of dependence on the cache signatures on the position of the ROP payload

Support Vector Machine (SVM) Approach

- A support vector machine algorithm can find a hyperplane in an N-dimensional space(N- no: features) that distinctly classifies the data points.

- We have implemented a linear S.V.M. classifier with four-fold cross-validation and calculated the accuracy of the classifier.

- The training, testing and validation datasets have equal distribution of vulnerable and non-vulnerable executions of the nginx binary.

Random Forest Classifier

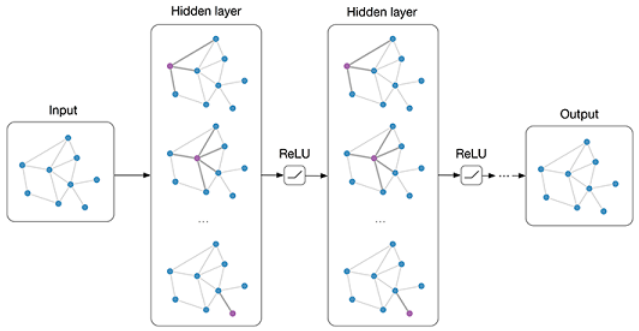
- Ensemble based classification
- Meta-estimator to fit a certain number of decision tree classifiers on different sub-samples of the data
- The accuracy is averaged to improve classification and address overfitting
- The sub-samples are selected from the original dataset with replacement (bootstrapping)

Introduction to Graph Neural Networks

- A graph is typically denoted as G(V,E) where V is the set of vertices and E is the set of edges connecting vertices in V.

- A Graph Neural Network operates on input data structured as a graph. It can process cycles, directed edges and relationships between nodes (and edges).

- A graph neural net converges to a stable equilibrium by the process of diffusion. All nodes in the neural net exchange information and update themselves accordingly until there are no more updates



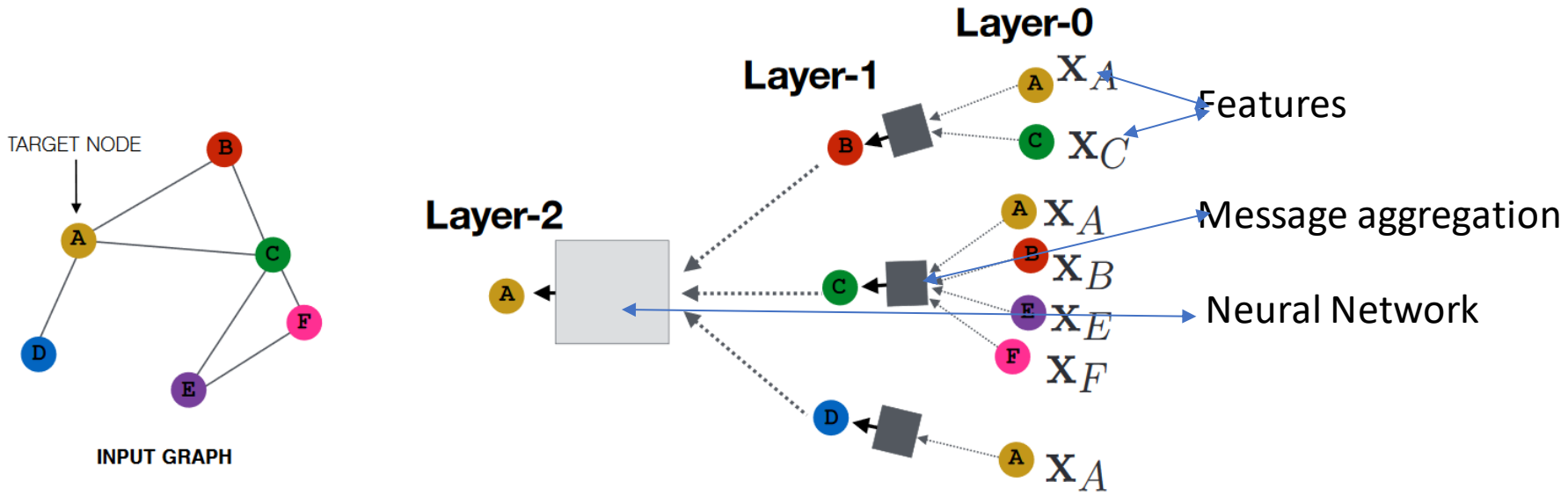
Src: Applications of GNN's (click on image)

Learning Framework

- Phase 1: Node embedding:-
  - Learning based on inverse of shortest path between 2 nodes
- Graph is input with a nxn adjacency matrix where n is the number of nodes and nxm feature matrix where m is the number of features
- Embedding vectors generated for each node
- Phase 2: Node classification with LSTM:-
  - Use node embedding and average binary features of the neighboring nodes using distance of learned embedding vectors

Node Embedding Generation

- Structural properties of the graph can be used for node embedding generation
- Here, we have used a neighborhood aggregation technique called nearest neighbors, based on unsupervised learning



Src: Stanfird tutorial

LSTM based Recurrent Neural Network

- We have designed a Long-Short Term Memory based Recurrent Neural Network (RNN)

- Our parameters to train the GNN for node embedding generation are
  - Nearest Neighbor based aggregation
  - Number of neighbors = 5

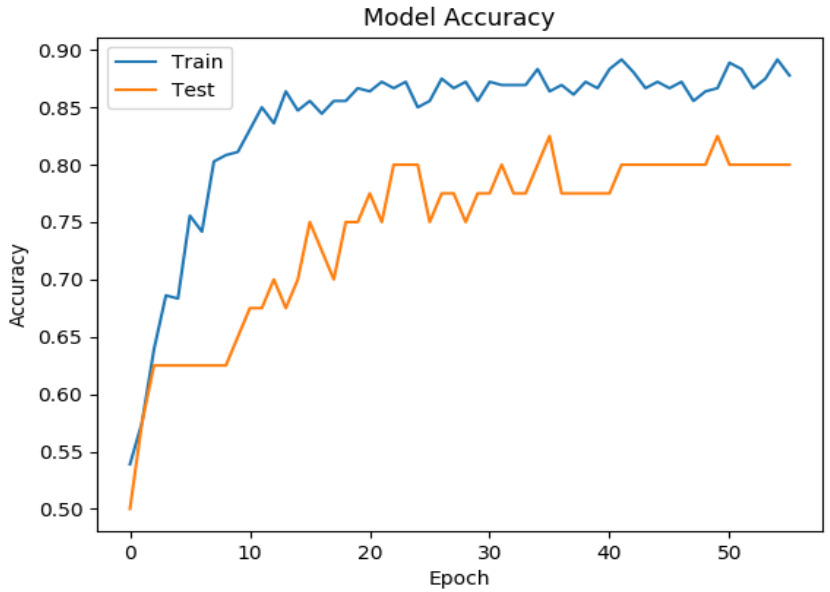
- Our parameters of the RNN are:
  - 2 layer unidirectional RNN
  - 512 hidden units
  - Initial Learning rate of 0.001
  - Learning rate decay of 0.0001
  - L1 regularization
  - 0.2 drop-out rate
  - Adam Optimizer
  - 30 epochs

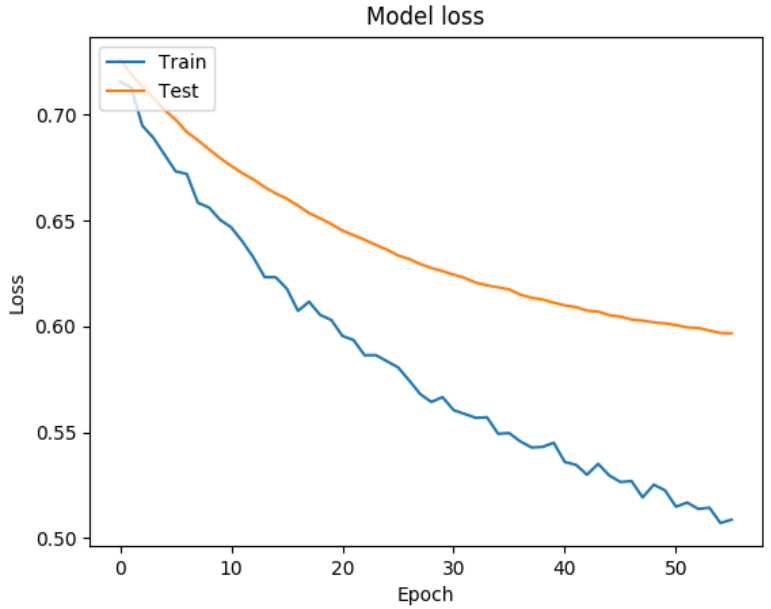
Results

- Accuracy = 91.25% (S.V.M.)
- Accuracy = 92.15% (Random Forest Classifier)
- Accuracy = 94% (GNN)
- Caveat: The high accuracy is due to the fact, that often features are the same across many data points when RoP payload is inserted in the early sections of the source code
- Solution: We have to run for more binaries, not just with nginx for diversified learning

Training Accuracy and Loss curve for GNN

- Overfitting is addressed by implementing early stopping for the same loss values
- Categorical cross-entropy loss function has been used





Conclusions

- We analyzed ROP attacks at hardware level
- We obtained cache statistics for programs running normally and under ROP attack
- Statistics show variations in L1 instruction cache misses
- The cache references, misses and miss rate can efficiently discriminate vulnerable and non-vulnerable executions
- We have received 90%+ accuracy with SVM classifier, Random Forest classifier and GNN based classifier
- More binaries like wireshark, httpd etc. have to be used for generalization in the learning by obtaining diverse data points

Thank You!

Contact:

Asmit De (asmit@psu.edu)
Saptarashmi Bandyopadhyay (szb754@psu.edu)