Lab 2b: Automated Malware Detection via Unsupervised Machine Learning and Binary Analysis

By: Malachi Jones, PhD

Outline

- Objectives
- Lab 2b Overview
- Lab 2b.1: Measuring Similarity of a Pair of Binaries
- Lab 2b.2: Prototype Malware Detector
- References

LAB 2B OBJECTIVES

Lab 2B Objectives

- After this lab, students should be able to
  - Apply the concepts and techniques developed via the lectures to challenging malware analysis problems by leveraging unsupervised machine learning and binary analysis
  - In particular, the ability to implement techniques to represent a binary as a set of n-grams that can be ingested into an unsupervised machine learning algorithm to identify anomalous (e.g. malware) binaries

LAB 2B OVERVIEW

Lab 2B Overview

- Develop a detection tool that can ingest a set of binaries and use clustering techniques to determine which binaries are “malicious”

- **Note:** *For the purposes of this lab, a malicious binary is defined as binary that is sufficiently dissimilar (< 85% similar) to other binaries such that it is not admitted into a cluster with other binaries*

- **Reminder:** ***No additional imports shall be added to any of the python files. Any additions will result in an automatic 0 for the portion of the lab***

Lab 2b Overview

- This lab will consist of the following sections
  - Lab 2b.1: **Measuring Binary Similarity (50% of Lab Grade)**
  - Lab 2b.2: **Building a Prototype Malware Detector (50% of Lab Grade)**

- **Note**: *Analysis will be performed on a set of 34 disassembled binaries in the lab 2b folder*

LAB 2B.1: MEASURING BINARY SIMILARITY

Lab 2b.1: Objectives

- **Lab 2b.1 Objectives**
  - Generate a set of n-gram hashes for a target binary, where the n-grams are composed of instruction mnemonics as discussed in the lecture
  - Implement a Jaccard similarity index that will then be used to compare the similarity of two test files
  - ***The resulting similarity of the two test files is approximately .2268***
  - ***Notes:***
    i. ***DisassemblyMnemonicNgramGenerator should support generating mnemonics for an arbitrary n.***
    ii. ***One point will be deducted if it can only support the default n=5.***

Lab 2b.1: Measuring binary Similarity

- **Lab 2b.1 Steps**

1. Generate a set of n-grams from a linear list of

```
def _generate_n_gram_set(self):

    """
    Generate a  set of n_grams from the linear_list_of_disassembly_instructions, where each element in the
    set is tuple of size n.
    Example: Suppose linear_list has 6 instructions: push, pop, mov, xor, add, sub.
             If n=3, then the first n-gram would be the following tuple: (push,pop,mov)
             The next n-gram would then be the following tuple: (pop,mov,xor)
             Then the next would be the following tuple: (mov,xor,add)


    :return: A set, where each element of the set is a n-gram represented as a tuple of size n
    """
    n_gram_set = set()

    logger.warning("@todo: Generate n gram set")

    """
    Hint 1:  Disassembly Instruction object has a property named mnemonic, which should be the type of the objects in
           a tuple. See the following code example
    """
    #================================Example relating to Hint 1================================
    disassembly_instruction = self._linear_list_of_disassembly_instructions[0]
    isinstance(disassembly_instruction, DisassemblyInstruction)
    mnemonic = disassembly_instruction.mnemonic
    logger.info("Hint 1: Mnemonic is '{}'".format(mnemonic))

    three_gram_tuple = tuple([self._linear_list_of_disassembly_instructions[0].mnemonic,
                              self._linear_list_of_disassembly_instructions[1].mnemonic,
                              self._linear_list_of_disassembly_instructions[2].mnemonic])

    logger.info("Example of a three gram tuple: '{}'".format(repr(three_gram_tuple)))
    #================================== END Hint 1 ==========================================

    return n_gram_set
```

The `_generate_n_gram_set()` method that will need to be updated is in the file `DisassemblyMnemonicNgramGenerator.py`

Lab 2b.1: Measuring binary Similarity (Step 1)

```
def _generate_n_gram_set(self):

    """
    Generate a  set of n_grams from the linear_list_of_disassembly_instructions, where each element in the
    set is tuple of size n.
    Example: Suppose linear_list has 6 instructions: push, pop, mov, xor, add, sub.
             If n=3, then the first n-gram would be the following tuple: (push,pop,mov)
             The next n-gram would then be the following tuple: (pop,mov,xor)
             Then the next would be the following tuple: (mov,xor,add)


    :return: A set, where each element of the set is a n-gram represented as a tuple of size n
    """
    n_gram_set = set()

    logger.warning("@todo: Generate n gram set")

    """
    Hint 1:  Disassembly Instruction object has a property named mnemonic, which should be the type of the objects in
           a tuple. See the following code example
    """
    #==================================Example relating to Hint 1==================================
    disassembly_instruction = self._linear_list_of_disassembly_instructions[0]
    isinstance(disassembly_instruction, DisassemblyInstruction)
    mnemonic = disassembly_instruction.mnemonic
    logger.info("Hint 1: Mnemonic is '{}'".format(mnemonic))

    three_gram_tuple = tuple([self._linear_list_of_disassembly_instructions[0].mnemonic,
                              self._linear_list_of_disassembly_instructions[1].mnemonic,
                              self._linear_list_of_disassembly_instructions[2].mnemonic])

    logger.info("Example of a three gram tuple: '{}'".format(repr(three_gram_tuple)))
    #========================================= END Hint 1 =========================================

    return n_gram_set
```

Important

Hint

Code Example

Lab 2b.1: Measuring binary Similarity

- **Lab 2b.1 Steps**

  2. Generate a set of hashed n-grams

```
def generate_set_of_hashed_n_grams(self):

    """
     Generate a hashed set of ngrams using mmh3 hash algorithm
    :return: hashed set of ngrams using mmh3 hash algorithm
    """

    n_grams_hashed_set = set()

    logger.warning("@todo: Generate a set of hashed n grams")


    """
    Hint 2:  Example for how to hash a tuple containing mnemonic objects
    """
    # ================================Example relating to Hint 2==============================

    three_gram_tuple = tuple([self._linear_list_of_disassembly_instructions[0].mnemonic,
                              self._linear_list_of_disassembly_instructions[1].mnemonic,
                              self._linear_list_of_disassembly_instructions[2].mnemonic])

    logger.info("Example of a three gram tuple: '{}'".format(repr(three_gram_tuple)))

    bytes_to_hash = b"".join([mnemonic.encode() for mnemonic in three_gram_tuple])
    hash_value = mmh3.hash(bytes_to_hash)
    logger.info("Hash value relating to Hint 2: 0x{0:02x}".format(hash_value))
    # ================================== END Hint 2 ========================================


    return n_grams_hashed_set
```

The `_generate_set_of_hashed_ n_gram_set()` method that will need to be updated is in the file `DisassemblyMnemonicNgramGenerator.py`

Lab 2b.1: Measuring binary Similarity (Step 2)

```
def generate_set_of_hashed_n_grams(self):

    """
     Generate a hashed set of ngrams using mmh3 hash algorithm
    :return: hashed set of ngrams using mmh3 hash algorithm
    """

    n_grams_hashed_set = set()

    logger.warning("@todo: Generate a set of hashed n grams")

    """
    Hint 2:  Example for how to hash a tuple containing mnemonic objects
    """
    # ==================================Example relating to Hint 2================================

    three_gram_tuple = tuple([self._linear_list_of_disassembly_instructions[0].mnemonic,
                              self._linear_list_of_disassembly_instructions[1].mnemonic,
                              self._linear_list_of_disassembly_instructions[2].mnemonic])

    logger.info("Example of a three gram tuple: '{}'".format(repr(three_gram_tuple)))

    bytes_to_hash = b"".join([mnemonic.encode() for mnemonic in three_gram_tuple])
    hash_value = mmh3.hash(bytes_to_hash)
    logger.info("Hash value relating to Hint 2: 0x{0:02x}".format(hash_value))
    # ====================================== END Hint 2 ==========================================


    return n_grams_hashed_set
```

Hint

Example Code related to hint

Lab 2b.1: Measuring binary Similarity

- **Lab 2b.1 Steps**
  3. Implement the Jaccard Similarity

```python
@staticmethod
def similarity(node_a, node_b):
    """
    Compute the jaccard similarity of the hashed_ngram_sets of Node A and Node B

    :param node_a: Node A
    :param node_b: Node B
    :return:
    """

    assert isinstance(node_a, ClusterNode), "Expected cluster node object"
    assert isinstance(node_b, ClusterNode), "Expected cluster node object"

    jaccard_similarity = 0

    logger.warning("@todo: Need to implement jaccard similarity")

    """
    Hint: Suppose we have two sets, Set A and Set B, the Jaccard similarity is expressed as the number of
          intersecting elements of the two sets diveded by the number of elements in the union of those sets
          In this context, the sets are the 'hashed_n_gram_set' property for each respective ClusterNode object
    """

    return jaccard_similarity
```

The `similarity()` method that will need to be updated is in the file `ApproxAgglomerativeClustering.py`

Lab 2b.1: Measuring binary Similarity (Step 3)

```
@staticmethod
def similarity(node_a, node_b):
    """
    Compute the jaccard similarity of the hashed_ngram_sets of Node A and Node B

    :param node_a: Node A
    :param node_b: Node B
    :return:
    """

    assert isinstance(node_a, ClusterNode), "Expected cluster node object"
    assert isinstance(node_b, ClusterNode), "Expected cluster node object"

    jaccard_similarity = 0

    logger.warning("@todo: Need to imp                    larity")

    """
    Hint: Suppose we have two sets, Set A and Set B, the Jaccard similarity is expressed as the number of
          intersecting elements of the two sets diveded by the number of elements in the union of those sets
          In this context, the sets are the 'hashed_n_gram_set' property for each respective ClusterNode object
    """

    return jaccard_similarity
```

Hint

Needs to be updated with the appropriate similarity score

Lab 2b.1: Measuring binary Similarity

- **Lab 2b.1 Steps**

  4. Execute the test harness to compute the similarity score of the pre-selected test binaries

```
def bin_diff(binary_a_pb_file_path, binary_b_pb_file_path):
    dis_mnemonic_gen_a = DisassemblyMnemonicNgramGenerator.from_disassembly_file(binary_a_pb_file_path)
    dis_mnemonic_gen_b = DisassemblyMnemonicNgramGenerator.from_disassembly_file(binary_b_pb_file_path)

    cluster_node_a = ClusterNode(dis_mnemonic_gen_a)
    cluster_node_b = ClusterNode(dis_mnemonic_gen_b)

    similarity = ClusterNode.similarity(cluster_node_a, cluster_node_b)

    return similarity


def test_harness():
    TEST_DISASSEMBLY_A_PB_FILE_PATH = "disassembly_protos/explorer.exe_Disassembly_70506db080603a6a35004e92edb2ed5bfa51fac9e0
    TEST_DISASSEMBLY_B_PB_FILE_PATH = "disassembly_protos/explorer_3416.exe_Disassembly_d5bc504277172be5c54b60ad5c13209dc1f72

    similarity = bin_diff(TEST_DISASSEMBLY_A_PB_FILE_PATH, TEST_DISASSEMBLY_B_PB_FILE_PATH)

    if round(similarity, 4) != .2268:
        logger.warning("Expecting a similarity score of approximately .2268.  Actual score is {}".format(similarity))

    logger.warning("@todo: Uncomment out the 'return' to proceed to clustering")
    return
```

No code needs to be modified at this point. Just execute the following python script: `ApproxAgglomerativeClustering.py`

Lab 2b.1: Measuring binary Similarity (Step 4)

bin_diff() method calls the appropriate methods to produce the similarity score

```
def bin_diff(binary_a_pb_file_path, binary_b_pb_file_path):
    dis_mnemonic_gen_a = DisassemblyMnemonicNgramGenerator.from_disassembly_file(binary_a_pb_file_path)
    dis_mnemonic_gen_b = DisassemblyMnemonicNgramGenerator.from_disassembly_file(binary_b_pb_file_path)

    cluster_node_a = ClusterNode(dis_mnemonic_gen_a)
    cluster_node_b = ClusterNode(dis_mnemonic_gen_b)

    similarity = ClusterNode.similarity(cluster_node_a, cluster_node_b)

    return similarity


def test_harness():
    TEST_DISASSEMBLY_A_PB_FILE_PATH = "disassembly_protos/explorer.exe_Disassembly_70506db080603a6a35004e92edb2ed5bfa5
    TEST_DISASSEMBLY_B_PB_FILE_PATH = "disassembly_protos/explorer_3416.exe_Disassembly_d5bc504277172be5c54b60ad5c1320

    similarity = bin_diff(TEST_DISASSEMBLY_A_PB_FILE_PATH, TEST_DISASSEMBLY_B_PB_FILE_PATH)

    if round(similarity, 4) != .2268:
        logger.warning("Expecting a similarity score of approximately .2268.  Actual score is {}".format(similarity))

    logger.warning("@todo: Uncomment o    e 'return' to proceed to clustering")
    return
```

Will produce a log warning message letting you know if the similarity score does not match the expected value of .2268

Lab 2b.1: Measuring binary Similarity

- **Submission**
  - You will submit a folder called lab_2b with the following contents
    i. ApproxAgglomerativeClustering
    ii. DisassemblyMenemonicNgramGenerator

  - **Note:** *Please do not submit any additional artifacts as they will not be evaluated*

Lab 2b.1: Measuring binary Similarity

- The relevant files for Lab 2b.1 (included in folder Lab2b) ***that will be updated*** are the following:
  - DiassemblyMnemonicNgramGeneratory.py
  - ApproxAgglomerativeClustering.py

- Files/Folders you may need to reference but ***should NOT be updated*** for Lab 2b.1
  - Disassemblies/*
  - disassembly_protos/*
  - Disassembly.py

LAB 2B.2: PROTOTYPE MALWARE DETECTOR

Lab 2b.2: Objectives

- **Objectives**
  - Execute the approximate clustering algorithm against the entire test data set
  - Identify the “malicious” binaries and log them as malicious by outputting their name and their hash

Lab 2b.2: Prototype malware Detector

- **Lab 2b.2 Steps**

1. Implement logic to identify the binaries that should be marked as malicious based on the definition of malicious we've defined in this lab

```
def perform_detection(self):

    # Initializing the agglomerative clustering object
    approx_clustering = ApproxAgglomerativeClustering(1-self._similarity_threshold)


    # Load the test dataset
    approx_clustering.load_binaries_from_directory(BINARY_FILE_PATH)

    # Perform the clustering
    approx_clustering.perform_clustering()

    # Get the cluster list
    cluster_list = approx_clustering.cluster_list
    # ==========================  HINT 1 ==================+==================

    """
    Hint 1: The below is a  example for code to perform the logging of a detected malware
            This example code should be commented out after you've successfully implemented the detection logic
            The binary that is referenced in this example may or may not be malware
    """
    cluster = cluster_list[0]
    cluster_node = (cluster.cluster_node_list[0])
    logger.info("(Malware Detected) Name:{}  Sha-256 Hash:{}".format(cluster_node.name,
                                                                    cluster_node.binary_sha256_hash))
    # ==========================END HINT 1 ==================================

    num_malware = 0

    logger.warning("@todo: Implement malware detection logic here")
    """
     Notes:  You will need to iterate through the cluster_list to determine which clusters meet the requirements
             specified in the lab for what "malware" is defined as
    """

    logger.info("total number of malware:{}".format(num_malware))
```

The `perform_detection()` method that will need to be updated is in the file `PrototypeMalwareDetector.py`

Lab 2B.2: Prototype malware Detector (Step 1)

```
def perform_detection(self):

    # Initializing the agglomerative clustering object
    approx_clustering = ApproxAgglomerativeClustering(1-self._similarity_threshold)


    # Load the test dataset
    approx_clustering.load_binaries_from_directory(BINARY_FILE_PATH)

    # Perform the clustering
    approx_clustering.perform_clustering()

    # Get the cluster list
    cluster_list = approx_clustering.cluster_list
    # ============================  HINT 1 ========================================

    """
    Hint 1: The below is a  example for code to perform the logging of a detected malware
            This example code should be commented out after you've successfully implemented the detection logic
            The binary that is referenced in this example may or may not be malware
    """
    cluster = cluster_list[0]
    cluster_node = (cluster.cluster_node_list[0])
    logger.info("(Malware Detected) Name:{}  Sha-256 Hash:{}".format(cluster_node.name,
                                                                    cluster_node.binary_sha256_hash))
    # ============================END HINT 1 ========================================

    num_malware = 0

    logger.warning("@todo: Implement malware detection logic here")
    """
     Notes:  You will need to iterate through the cluster_list to determine which clusters meet the requirements
             specified in the lab for what "malware" is defined as
    """

    logger.info("total number of malware:{}".format(num_malware))
```

Hint

Code example for logging detected malware

Important

Lab 2b.2: Prototype malware Detector

- **Lab 2b.2 Steps**
  2. Execute the PrototypeMalwareDetector script and observer the output.

```
INFO:PrototypeMalwareDetector:(Malware Detected) Name:explorer.exe  Sha-256 Hash:b'70506db080603a6a35004e92edb2ed5bfa51fac9e0
INFO:PrototypeMalwareDetector:(Malware Detected) Name:POWERPNT_11980.EXE  Sha-256 Hash:b'7201101dcd62724937c4f7a4476176ea0da6
INFO:PrototypeMalwareDetector:(Malware Detected) Name:calc.exe  Sha-256 Hash:b'c74f41325775de4777000161a057342cc57a04e8b7be17
INFO:PrototypeMalwareDetector:(Malware Detected) Name:iexplore.exe  Sha-256 Hash:b'70c9616c026266bb3a1213bcc50e3a9a24238703fb
INFO:PrototypeMalwareDetector:(Malware Detected) Name:executable.rundll32.exe_4084.exe  Sha-256 Hash:b'1ce0bbdaa5a1a9eb51b514
INFO:PrototypeMalwareDetector:(Malware Detected) Name:POWERPNT.EXE  Sha-256 Hash:b'63d995dd8f5b82f96e3f0b76a0acf1b4e4a8b7c8f3
```

Note: *The above is a small sample of the actual logging message and **not the complete list** of binaries that should be flagged as malware*

Lab 2b.2: Prototype malware Detector

- **Submission**
  - You will submit a folder called lab_2b with the following contents
    i. PrototypeMalwareDetector.py

  - **Note:** *Please do not submit any additional artifacts as they will not be evaluated*

Lab 2b.2: Prototype malware Detector

- The relevant files for Lab 2b.2 (included in folder Lab2b) ***that will be updated*** are the following:
  - PrototypeMalwareDetector.py

- Files/Folders you may need to reference but ***should NOT be updated***
  - Disassemblies/*
  - disassembly_protos/*
  - Disassembly.py
  - DiassemblyMnemonicNgramGeneratory.py
  - ApproxAgglomerativeClustering.py

References

1. Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, *19*(4), 639-668.