Binscan Binary Utility

Anthony Frazier
ENEE459B Fall 2020

Introduction

Binscan is a binary utility used for analyzing ELF-64 objects. A variety of different aspects of an ELF-64 object file are printed, including

1. The **SHA1 checksum** of the text section of the binary
2. All **unique x86 instructions utilized and the frequency of their use**
3. The **Renyi Entropy of the ELF-64 object's text section**
4. The **SHA256 checksum** of the text section of the binary

When the binscan utility is run from the command line, these aspects are printed to STDOUT. A binary is also created named "file.bin" where file is the ELF-64 object passed into the analyze. The user may later open these bin files with the -open option.

Note: Before retrieving the information from the bin file, the user must enter a valid password. For testing purposes, the admin password of the binary is "notthepassword". The standard user password of the binary open function is "letmein". The admin password is stored within openscan.h, with the user password being stored within the main function of openscan.c. ***This is one vulnerability present within the binary utility. An attacker could retrieve the admin password by examining the contents of the header compiling the utility or gain user access through disassembly of openscan.c.***

Installation

1. Install the OpenSSL and Capstone C packages.
2. Extract the binscan directory to your desired installation location.
3. Make sure all header files are contained within the folder bin/.
4. Use **make** to compile the Binscan executable in the main binscan directory.

SHA Checksums

The SHA1 checksum of the text section of the binary is calculated using OpenSSL's C library on individual bytes within the binary's text section.

Note: In order to access the text section of the binary for this calculation and later calculations, the libelf library is utilized to retrieve a data buffer pointer from the text section's header. One may traverse the sections until the desired section name is found, and then manipulate the section as they see fit. For more information on how this was accomplished, please refer to chapter 5 of Libelf by Example.

The SHA1 checksum is stored within a SHA1Record struct defined in *binproto.h*.

```
typedef struct{
  EntryType et;
  uint8_t sha1[SHA_DIGEST_LENGTH];
} SHA1Record;
```

This structure contains an entry type for the file-header structure declared above and an array of bytes the size of the SHA1 digest length declared in openssl/sha.h. Below is a sample output of the SHA1 checksum performed on a simple binary.

```
afraz98@enee459b-1:~/proj1$ ./binscan -analyze helloworld
SHA1: bea802b9286a756dd01bfdc316044dd07b55bb37
```

A similar output can be found in the bin file generated by the analysis for the same binary once the SHA1Record structure is retrieved from memory.

The SHA256Record differs in the size of the byte array due to differences in the two hash functions. They are otherwise both implemented with the OpenSSL C library with some minor deviations in library functions utilized.

Printing x86 Instructions

The Binscan binary utility prints each unique instruction with the amount of times they are called after disassembly is performed by the Capstone disassembly library. A pointer to the data buffer of the text section is provided to this library, and instructions are stored within an instruction buffer structure declared within *binproto.h.*

```
typedef struct{
  Instruction instructions[MAX_INSTRUCTIONS];
  int ninstructions;
} IBuffer;
```

This structure contains an array of Instruction structures and the number of instructions present within the text section.

```
typedef struct{
  char instruction[LONGEST_OPCODE];
  int instruction_calls;
} Instruction;
```

The instruction structure pictured above consists of an opcode string and the number of calls to said instruction. Again, this definition may be found in *binproto.h.*
Below is an example output of the instruction analysis:

```
115 instructions

xor     3
mov     20
pop     15
and     2
push    14
lea     11
call    7
hlt     2
nop     15
cmp     5
je      7
test    4
jmp     4
ret     8
sub     4
sar     4
shr     2
add     4
jne     3
```

Renyi Entropy

The Renyi Entropy of the text section is calculated as follows:

$$H_2 = -log_{256}(\sum p_i^2)$$

Where p_i^2 is the squared relative frequency of each byte stored in memory. The *math.h* library file provided the functions necessary to perform the logarithm required for this value. This value is stored within a RenyiEntropy structure containing a double value representing this calculation.

This value represents a measure of compression or encryption within a specified binary. In static file analysis, a lower value of this entropy would signify that decompression or decryption would be less likely to be necessary for analysis.

To calculate this value, all unique bytes within the text section are found and inserted into an array. The relative frequencies of these bytes are found, squared, and summed together. Next, the logarithm is calculated.

Utility Anatomy

The following is a system-level description of the software components enabling the binary to function. To better understand the components of the software, it is advised to obtain a reference copy of the software for inspection.

The Binscan utility is split into three sections, translating to four object files:

1. ELF-64 Object Parsing
2. Command-Line Interface Handling
3. Binary Analysis File Parsing
4. Binary Analysis File Encryption and Decryption

These sections may be translated to *libelf64.o, binscan.o, openscan.o / compare.o, and encrypt.o* respectively.

The *compare.o* object contains an x86 function testing if two character inputs are the same. This method is used to verify that the user has entered the correct passwords associated with the two use modes.

Command Line Interface Handling

The command-line interfacing portion of this utility is implemented within *binscan.c* mapping to *binscan.o*. This function serves as a primary driver between the ELF-64 object parsing and binary file parsing based on user input. Below are the function prototypes for relevant functions:

```
parseElf(char *file); //Begins ELF-64 object analysis
openAnalysis(char *file); //Opens binary analysis file created by analysis
printHelp(); //Prints usage information to command line for user
```

This object file primarily servers as a user driver allowing the user to enter the two modes implemented for the utility. For more information on how these two modes operate, please refer to the sections below.

ELF-64 Object Parsing

The ELF-64 object parsing portion of the utility maps from *libelf64.c to libelf64.o.* The OpenSSL, Capstone, and Libelf libraries are used within this object file to retrieve the necessary information for binary analysis. Below are some relevant function prototypes:

//Create ELF-64 object for analysis
Elf *openELF(char *file, int fd);

//Check if object is ELF-64
int checkElf64(Elf *e);

//Use section header to find text section
GElf_Shdr findTextSection(Elf *e, Elf_Scn **s);

//Print SHA1 checksum utilizing OpenSSL
void printSHA1(Elf *e, byte *sha_value);

//Print SHA256 checksum utilizing OpenSSL
SHA256Record printSHA256(Elf *e);

//Parse text section for data buffer, call print instructions
IBuffer parseSectionText(Elf *e); /

//Print unique instructions with call counts with Capstone library
IBuffer printInstructions(unsigned char* buffer, size_t buffersize, uint64_t address);

//Calculate Renyi entropy from text section bytes
RenyiEntropy calculateEntropy(Elf *e);

//Output SHA1 and FileHeader to file
(Note: Later structures written to file from *parseElf())*
int fillFileBuffer(uint8_t *buffer, char *argfile, uint8_t *sha1);

Binary Analysis File Parsing

The binary analysis file parsing is accomplished by *openscan.c* mapping to *openscan.o.* The password entered is first checked with the user and admin passwords before the analysis file is parsed. Below are some relevant function prototypes:

//To be called by binscan driver
void openAnalysis(char *file);

//Perform analysis as admin
void analysisAdmin(char *file);

//Perform analysis as user
void analysisUser(char *file);

//NASM Assembly function comparing characters
extern long compareCharacters(char a, char b);

The analysis retrieval is accomplished by using *fread* with local struct variables according to the data types listed in *binproto.h*. Once these data types are retrieved, they are manipulated and printed to STDOUT.

Binary Analysis File Encryption and Decryption

The binary analysis files are encrypted via a simple shift cipher operating on the individual bits of the analysis file. Relevant functions are implemented in *encrypt.c* mapping to the *encrypt.o* object file. Below are the relevant function prototypes:

//Encrypt binary analysis file
void encryptFile(char *file, char *outputfile);

//Decrypt binary analysis file
void decryptFile(char *file, char *outputfile);

Usage Instructions

Binary Analysis

To use this binary tool, call it from the Linux bash shell with the following arguments:

./binscan -analyze <binary>

Where <binary> is a valid ELF-64 object. After the analysis is printed to STDOUT, a file named <*binary*>*.bin* will be placed within the current directory.

Open Analysis Files

To access analysis files, input the following:

./binscan -open <binary>.bin

You will be prompted for the appropriate user or admin password. After success authentication, the utility should print a similar analysis to STDOUT.

Analysis File Encryption

The Binscan utility has an encryption function preventing analysis files from being accessed should some malicious user enter the system. To use this tool, call the utility as follows:

./binscan -encrypt <binary>.bin <outfile>.bin

Where <*outfile*>*.bin* is the desired encrypted file name.
Note: Be sure to delete the version of the analysis file that is not encrypted!

Before attempting to analyze an encrypted file, the user should call the utility as follows:

./binscan -decrypt <outfile>.bin <analyze>.bin

Where <*analyze*>*.bin* is the desired name of the analysis file to be analyzed.