Image-based Ransomware

Justin Garrigus
Computer Science Department
University of North Texas
Denton, Texas, USA
justingarrigus@my.unt.edu

***Abstract*—Ransomware applications present a threat to the availability of user data by encrypting files after a malicious program is executed, only decrypting the files after some payment is received. This malicious program may be introduced to the user through non-standard means through exploitation of obscure attack surfaces. In this work, we show a new ransomware that is based on images: the ransomware targets a toy image-viewing application and infects the user with a buffer-overflow introduced through a malformed image, and it encrypts the users files with RSA and transforms the result into an image the user can view. Key distribution is based on a server/client symmetric/asymmetric encryption technique that ensures users cannot retrieve their files without communication with the malicious server owning the keys. Additionally, a file-monitor is included that checks the state of the system for read/write accesses and notifies the user when mass-encryption appears to occur.**

This project places a large emphasis on low-level execution details and includes a manual implementation of RSA encryption/decryption with arbitrary-size keys. The resulting program is not meant to be high-end or otherwise unbreakable, but is instead a learning exercise to encourage an understanding of how cryptography primitives work.

***Index Terms*—ransomware, cryptography, RSA, encryption**

I. Introduction

Ransomware presents a combination of many distinct areas in computer security, including parallel processing, operating system and hardware design, complexity analysis, networking, and cryptography. Understanding how ransomware works by building a new malicious program gives an opportunity to combine these areas together and can be safely done through utilization of a virtual environment. Some techniques that are popular in ransomware like key encryption and buffer overflows are necessary for all computer scientists to understand regardless of discipline or specialization.

This project is an implementation of the standard approach ransomware applications apply to encrypt files but with novel concepts that increase its originality. Some of these concepts are for pedagogical purposes at the expense of decreasing the security of the program, while others do not affect the programs security.

Typical ransomware applications are divided into several parts:

1) Infection: the malware is received by the user and modifies the behavior of their system.
2) Encryption: files on the user's system are modified to the extent it would be reasonably difficult for a dedicated user to reverse.
3) Decryption: after payment is received, the attacker communicates some method to the user on how to return their files to the original state.

The variability in ransomware techniques stems from the difficulty of the infection step: it is exceedingly difficult to get a user to run arbitrary code on their system due to the security practices software engineers employ in their operating systems and antivirus programs. Although, certain cybersecurity concepts remain prevalent in the distribution of malware despite their age and widespread popularity. For example, buffer overflows are common due to their ability to place data into executable regions of memory, and other approaches like heartbleed are variations on the standard method.

This paper outlines our approach at creating a new ransomware based around images. It assumes the user has an image-viewing software with a vulnerability in it that allows a malformed image file to write create log files with new data and file names, letting an attacker overwrite an executable file with new code. The attacker can then simply send the user an image file through social media or other communication means and wait for the user to attempt to open it. Once infected, the user unwillingly generates a public-private key pair, has their private key encrypted by the attacker, and uses their public key to encrypt their own files. To decrypt their files, the user would send payment along with their encrypted private key to the server which the server would decrypt and then return. Finally, the user can re-execute the malware with the private key, returning the files to their original state.

In contrast to this standard approach to ransomware, several new features were added. First, the encryption method is a C implementation of RSA with arbitrary-precision arithmetic, which includes a Miller-Rabin prime number finder for the keypair generator. Second, after the user's files are encrypted with RSA, they are transformed into image files that can be viewed by other software. The images are encoded with least significant bit (LSB) steganography which takes a base image (either requested from the malicious server or taken from an online website) and encodes a given file directly into it. The result of this is a unique attack method that the user can actually see the results of: the user can open up a sensitive file (e.g., one containing a list of important passwords) and view the contents of it in a new form (e.g., changing this ”passwords.txt” file into a picture of a dog).

This paper is divided into four sections. Section II gives an overview of the theory of ransomware with explanations on the cryptography and practical details. Section III details the implementation of the system itself and their component parts, including the user system, infection, key creation and distribution, image transformation, and monitoring, along with potential modifications that could be made to the design. Section IV gives a step-by-step walk-through of the actions the ransomware takes from initial infection to removal.

II. Related Works

This section will describe the background to the ransomware. Since a focus on the project was on recreating components at a low level, emphasis was placed on creating both a modular design and a theoretically-sound product. In particular, three components required researching: steganography, RSA encryption, and buffer overflows.

A. Steganography

Steganography is the act of hiding data within a file without a casual observer being able to notice the original file was modified. The basis of our design was described in [1], which outlines several methods of increasing complexity on how to encode information within an image. The simplest method is to use least significant bit encoding, which takes a message of size n bits and encodes k bits per byte in the image, shown in figure 1. Assuming a three-channel color image, the least significant bits are the least important in conveying the contents of the image. As such, they are the easiest to replace without the end user noticing something was changed. If $k = 1$, then the encoded message is invisible to any human, and is easily mistaken for compression.

Fig. 1. LSB steganography, with (left) the base image and (right) one bit of the ”lorem ipsum” sample text encoded in each byte of the image.

[1] also describes additional methods of steganography besides the basic LSB encoding scheme including use of a Caesar cipher and key-based columnar transposition, but these methods are unnecessary for this project. In fact, steganography is typically used to be undetectable by the end user, but this is not a particular concern for a ransomware application; ransomware developers want the end user to act quickly after learning about the encryption of their files in order to encourage impulse decisions on sending payment, and users may not notice–or otherwise may not gain any sense of urgency–if they do not immediately realize important data was lost. This means we can afford larger k values for our steganography approach, which lets us encode more bits of the message into the base image per byte.

B. RSA encryption

The encryption primitive employed for this project was RSA. This was originally chosen due to its simplicity and popularity as well as it being asymmetrical, which makes it easier to hide an encryption key within the malware without concern of the user finding a way to decrypt their files on their own. The RSA algorithm consists of a prime-number generator and key creation.

The goal of the prime number generator is to quickly find two prime numbers of sufficiently-large length that can be used for key generation. The issue is that deterministic primality tests are extremely time consuming especially for secure key sizes (e.g., 2048-bit keys which require two prime numbers 1024-bits long), so the nondeterministic Miller-Rabin primality test [2] is a sufficient alternative to give numbers with high a likelihood of being prime.

In plain English, the Miller-Rabin test randomly chooses a large number and sequentially tests each of their successors for primality. The Prime Number Theorem [3] approximates the amount of prime numbers that exist below some threshold, and it predicts that the likelihood p of a given number being prime decreases with the number of digits d, calculated with the expression

$$p = \frac{1}{d-1}$$

For example, if a 64-digit number is chosen at random, the probability of it being prime is 0.0159. Put another way, it would take on average 63 primality tests until we identified a prime number.

Deterministic tests are generally much slower or require much more pre-computed information than probabilistic tests, but have the benefit of guaranteeing if a number is prime. RSA takes the product of two prime numbers $N = p \times q$ and relies on the difficulty of factoring N into their given components. If p or q turned out to be composite, then the product N would be easier to factorize, which would undermine RSA's security. This may make probabilistic tests like Miller-Rabin unattractive since it is unable to guarantee a number is prime, but an extra mechanism is included in Miller-Rabin to include their confidence: repeated tests.

Miller-Rabin can be considered more as a ”compositeness” test rather than a primality test due to its ability to confidently tell if a number is composite but only assume when a number is prime. The way it checks this is by running k iterations of their composite test, where each test essentially has a 25% chance of falsely announcing a composite number is prime. As such, the probability of a composite number being assumed as prime is 4^{-k}. This means a high enough k value is nearly guaranteed to yield correct prime numbers. A popular iteration count is $k = 40$, which is high enough to ensure that an unavoidable hardware failure is more likely to result in a false-positive than the primality test itself is.

C. Buffer Overflow

Buffer overflow attacks exploit a flaw in program design in which critical data is placed in the same address space as user-writable data. The critical section is illegally modified by the user which changes the output of the program in some unintentional way. The standard form of buffer overflow [5] is a stack-based ”stack-smashing attack” which works by copying the data from a user-writable portion of memory into a buffer of limited size, specifically overwriting the portion of the stack that holds the return address of a function. The program would then jump to the code specified by the address which may contain more arbitrary data the user specified themselves, allowing them to grant a new process escalated privileges or to otherwise view or edit data the process was not otherwise intended to observe.

This stack-smashing attack is well studied but difficult to setup. It requires very detailed information about a target program and careful consideration of runtime information–which may be gained from observation of the source code or binary executable file–but still can only be performed once certain guards like a non-executable stack and StackGuard canaries [5] are explicitly removed from the compiled program.

An alternative to this approach is to create a buffer overflow method that works regardless of compiler flags being set. This would require a software-only approach to overflowing data only in user-writable portions of memory, placing executable data in places the program already expects executable data to be in. A simple example of this attack is shown in listing 1, in which a flag `ADMIN_FLAG` exists as a predefined macro, but is overwritten if the value in the string `name` is greater than the size of the buffer. Any name that is 7 characters or longer will cause `flag` to contain a nonzero value, which makes the condition at line 8 evaluate to `true` and the subprocess to be spawned with escalated privileges.

```
1  void spawn_subprocess(char* name) {
2      char buffer[6];
3      // Macro: 0 if low privilege,
4      // 1 if high privilege.
5      int flag = ADMIN_FLAG;
6
7      strcpy(buffer, name);
8      if (strcmp(buffer, "admin") == 0 || flag)
9          system("sudo ./run_process");
10     else
11         system("./run_process");
12 }
```

Listing 1. Simple buffer overflow attacks

A variation of the attack with user-defined code is shown in section III, but the concept remains the same: no additional compiler flags can prevent this attack because it relies on the software implementation’s insecurity and inherent language requirement that C does not check for array bounds when accessing data. As such, this attack is much more portable and can be executed on a larger number of systems.

III. Approach

This section will describe each of the core components for the ransomware application. The ransomware builds on the aforementioned related works by integrating them into different C programs combined together to make a single malicious application. This includes the cryptography module, the buffer overflow module, the steganography module, and the monitoring module.

A. *Cryptography*

The cryptography code builds off of the RSA algorithm with the Miller-Rabin prime number generator. This section is inherently less secure than standardized cryptography systems like the one given in `openssl` due to it being a simplified implementation for pedagogical purposes, but it still functions regardless as a low-security encoding device that would be difficult to crack without experience in cryptography.

In particular, RSA is normally a deterministic encryption scheme which gives a consistent output for the same key and input sequence. This can be insecure due to plaintext attacks, where unencrypted messages are passed through the public key in order to test if they match the encrypted data. This could cause the victim to avoid paying for the decryption key by attempting to rebuild their important files by generating likely candidates for the files, encrypting it with the public key, comparing it to the ciphertext, and accepting it as the original file if it matches. Padding schemes like PKCS#1 [6] avoid this by essentially randomizing the input plaintext before it is passed to the RSA encryption method which makes it considerably more difficult to brute-force decode files without the private key. Although, padding was not implemented in this project due to time constraints, but it can easily be added by simply transforming the input message before it is piped to the RSA component described next.

Listing 2 shows the main overview of the Miller-Rabin method to test if `n` is prime. First, the method picks a number $n - 1$ and divides it until it gets an odd number (2-4), which is synonymous with removing the trailing 0’s from `n`’s binary representation. Then, it `composite_test`’s the number `ITERS` times (5-6), only announcing the number as prime if it passes every test (8). This `ITERS` value is the ”iteration count” mentioned in section II, where each iteration has at most a 75% chance of correctly confirming `n` as composite–or, alternatively, falsely suggesting `n` is prime when it really is not.

```
1  int millerrabin(int n) {
2      int d = n - 1;
3      while (d is even)
4          d = d / 2;
5      for (int i = 0; i < ITERS; i++)
6          if (composite_test(n, d) failed)
7              return 0;
8      return 1;
9  }
```

Listing 2. Miller-Rabin primality test

The `test` method in listing 2 is the compositeness test that checks if the `n` is composite, which is summarized in listing 3. The mathematical workings of this function is out of scope for this paper, but the important part is the random-number generator (2) which makes repeated executions of `composite_test` yield different results. Additionally, it is important to note that this test does not confirm that a number is prime but rather announces if different congruence relations

(4, 9) are true. The value `a` in this case is checked if it is coprime to `n`, or `GCD(a, n) == 1`, which means if a random variable `a` has a non-zero factor shared with `n`, then `n` therefore has a factor greater than one, which makes it non-prime.

```c
int composite_test(int d, int n) {
    int a = random(2, n-2); // Inclusive
    int x = pow(a, d) mod n;
    if (x == 1 || x == n - 1)
        return 1; // "n" may still be prime
    while (d != n - 2) {
        x = pow(x, 2) mod n;
        d = d * 2;
        if (x == 1)
            return 0; // "n" is definitely composite
        else if (x == n - 2)
            return 1; // "n" may still be prime
    }
}
```

Listing 3. Miller-Rabin compositeness test

The Miller-Rabin test can be rewritten to generate prime numbers easily. Noting from section II that the probability of a number with d digits being prime is approximately $\frac{1}{d-1}$ due to the prime number theorem [3], we can choose some initial value n that is d digits long and subsequently test itself and approximately $d-2$ of its successors for primality with the Miller-Rabin test shown in listings 2 and 3. This now completes the backbone of the RSA algorithm shown next.

The RSA algorithm used in this project has three core components: key-pair generation, encryption, and decryption. Key-pair generation is shown in listing 4, represented as a function that yields an `n`, `e`, and `d` value. It works by first creating two large prime numbers (2-3), getting their product `n` (4), calculating the Carmichael's totient function `l` (5), and findingthe modular multiplicative inverse of a constant value `e` (6-7). The two values `n` and `e` make up the public key for encoding plaintext into ciphertext, and the values `n` and `d` make up the private key for decoding ciphertext back into plaintext.

```c
1 tuple(n, e, d) generate_keypair() {
2     int p = generate_prime(); // Miller-Rabin
3     int q = generate_prime();
4     int n = p * q;
5     int l = lcm(p-1, q-1);
6     int e = 65537;
7     int d = pow(e, -1) mod l;
8     return tuple(n, e, d);
9 }
```

Listing 4. RSA key-pair generation

Once a key-pair is generated, the private key is kept hidden by the user and the public key is distributed to anybody who needs to send a secure message to the user. Encryption is performed by splitting a message into `b`-byte chunks, where `b` is the size in bytes of the value `n` in the key-pair, and transforming each chunk c_{plain} into a new encrypted number c_{enc} through the equation $c_{enc} = (c_{plain})^e \mod n$. Each chunk can be stored into a different file or transferred over a network. The reverse process, or returning an encrypted chunk back into its plaintext form, can be done with the equation $c_{plain} = (c_{enc})^d \mod n$.

An important detail to note is that integers in a language like C are fixed-length of size less than 8 bytes long, and RSA prime keys need to be hundreds or even thousands of bytes long. This requires use of an arbitrary-precision arithmetic module and we utilized the GNU Multiple Precision Arithmetic Library [7] to do this[1].

B. Buffer Overflow

The buffer overflow technique used for this project is similar in concept to the one shown in section II. It relies on the fact that a block of arbitrary data writable by the user can become executable if it is placed into an area of memory the user has previously denoted as executable. This program works by parsing `ppm` image files which have three important data fields: an image `width`, `height`, and buffer of RGB binary data of size `width * height * 3`. The program may contain a struct like the one shown in listing 5 to hold this data.

```c
struct imgdata {
    int width, height;
    char *buffer;
    char *logfile;
    char *logname;
    int  *logfile_size;
};
```

Listing 5. Struct with a buffer-overflow vulnerability

```c
1  void parse_image(char* file_name) {
2      FILE *image_file = open file_name for reading;
3      data->width, height = read from image_file;
4
5      struct imgdata *data = allocate buffers;
6      strcpy(data->logfile, "Successfully opened.");
7      strcpy(data->logfile_name, "log.txt");
8      data->logfile_size = strlen(data->logfile);
9
10     data->buffer = char_copy(image_file);
11
12     save data->logfile to data->logfile_name;
13 }
```

Listing 6. Image-parsing application with buffer-overflow vulnerability

An overview of our program is depicted in listing 6. First, we assume that `buffer`, `logfile`, `logname`, and `logfile_size` in listing 5 each exist in the same portion of memory, which can be achieved by using `malloc` to allocate a large block of memory and assigning each pointer to a different location of that block (5). In our program, we assume that a file with the name of `logname` is written at the end of execution with the contents of `logfile` (12). Once the file is successfully opened, we intend to write the string `"Successfully opened."` to a log file named `log.txt` (6-7). Then, the program proceeds to read the

[1]The GMP library has a function `mpz_probab_prime_p` that automatically performs the Miller-Rabin test and a function `mpz_nextprime` that automatically identifies the next prime number greater than some threshold value, but neither of these functions were used in this project in order to create a new solution from scratch for pedagogical purposes.

contents of the image file itself and copy the data into the image buffer (10).

The issue with the code in listing 6 is if the `image_file` contains malformed data. Consider a situation in which the size of the buffer is allocated (3) before copying the buffer itself (10). In most programs, the amount of data read from the file would equal to the expected size of the buffer (e.g., by utilizing a method like `strncpy`), but an insecure program may utilize a method like the one shown in listing 7 which copies a buffer regardless of the expected file size. Properly-formed image files would have no issue with this design, but a malicious user could create an image with the following traits:

1) The `width` and `height` of the image are each set to 0. The program will use `char_copy` (listing 7) or some other method to copy the rest of the data in the file into the struct shown in listing 5.
2) The buffer segment of the image file instead contains the data copied from a malicious executable file. This overwrites the `logfile` variable in the struct, which is the contents of the log file.
3) After the executable file, the name of the file is set to the name of some executable file the attacker is sure the user will run in the future. In our program, we used `"view_image"`, which is a hypothetical image-viewing software which is run after the `"parse_image"` program finishes. This string overwrites the `logname` variable in the struct, which is the name of the log file.
4) Finally, the size of the executable file is written after the new name of the log file. This overwrites the `logfile_size` variable.

```c
void char_copy(unsigned char* buffer, FILE* f) {
    unsigned char ch;
    unsigned char *buffer_ptr = buffer;
    for (int written; written != 0;) {
        written = fread(&ch, 1, 1, f);
        *buffer_ptr = ch;
        buffer_ptr++;
    }
}
```

Listing 7. A common programming pattern typically used in debug code to read every byte from a file

Typically, the program will write an innocuous log message as shown in listing 6 to the file `log.txt`, but the attacker can modify this to point to an executable file on the user's machine (for example, `/usr/bin/bash`). The attacker can construct some malicious program and then copy the contents of that program into their image file so it gets copied into this log file location. If the log file name is the same as an executable file on the user's system, then this allows arbitrary code to get executed.

C. Steganography

The steganography module is an optional portion of the ransomware that encodes a message into an image with LSB encoding. This does not affect the level of security for the ransomware due to the simple encoding scheme and widespread knowledge of the LSB method of encoding, so a dedicated user with lots of resources would be easily capable of returning their files to their unencrypted state if this was the only defence. That being said, in the context of a real-world situation, image-based encryption would have the added benefit of creating a novel, distinct-looking ransomware. If popularity is a concern for ransomware developers, then a program which transforms arbitrary files into recognizable images would be favorable as the news of it would spread more quickly than a standard ransomware would.

A concern presented in section II was the limited size of LSB encoding. The typical use of this method is to communicate messages in data without the end user discovering a message was encoded into the file, but this is not a concern for a ransomware. As such, we can adjust the standard method by increasing the number of bits encoded into the image. Figure 2 shows the result of LSB steganography being performed with four bits of encoded message for each byte of the image, as opposed to 1 showing only one bit per byte. Even with this much data being encoded, the resulting image still looks almost identical to the base image. Enlarging the image shows artifacts that looks like compression or a high-contrast filter, but this still can be easily mistaken for a blurry image rather than a deliberate attempt at data encoding.

D. Monitor

The monitor is a defense against the ransomware attack and can be used as a precautionary method to prevent a ransomware from spreading to larger portions of the operating system. It works by checking a directory for a typical ransomware signature and sending a notification to the system when one occurs. This project utilizes the `inotifywait` module to handle automatic updates when a directory is modified.

In particular, the specific events that are monitored are `access`, `create`, and `delete`. Ransomware–or at least the one described in this project–function by reading a file for its contents (`access`), passing the stream of data from a file through a transformation function to yield a new encrypted file (`create`), and deleting the original unencrypted file (`delete`).

The monitor takes the form of an `inotifywait` command that observes a given directory and outputs the event to `stdout` when one occurs. The event then contains the file name, directory name, and event type that occurred. We created a python script which will run on the host's system and parse each modification to the directory by checking the `stdin` connected to the `inotifywait`'s output.

The monitoring process assumes that the access, create, and delete events happen in quick succession from each other. Ransomware need to encrypt lots of information very quickly to maximize the damage performed, so it is not unreasonable to assume that the events are temporally localized. Furthermore, an end user may realize something bad is occurring if the size of available memory on their system decreases very quickly, so ransomware may be unable to avoid our detection method by, for instance, bulk-deleting all the original files at the very end of the encryption process instead of deleting the files as they are encrypted.

The monitor creates a queue and organizes the events in terms of their execution times. Events that occurred more than a `time_threshold` amount of time ago are removed from the queue; for our program, a threshold value of about 10 seconds is more than enough time to detect if a ransomware attack is being performed. Next, the monitor correlates two types of events together:

1) We connect the `access` and `delete` events of the same file together. If this occurs, then an `accdel_counter` value increments.
2) We note the number of times that files are created by incrementing another `create_counter` value.

When an event occurs more than the `time_threshold` time ago, then the two counters are decremented back in order to "undo" the changes the event caused.

Finally, whenever an event is spawned, the two counter values are compared. If both `create_counter` is within $0.8\times$ of `accdel_counter`, and `accdel_counter` is greater than a `accdel_threshold` value (which we set to be equal to $\min[10, 0.8 \times \texttt{len(directory)}]$), then a ransomware attack is assumed to be taken.[2] The system administrator can respond to this however they choose (either by suspending all processes and backing up the specific directory or by killing the processes that are causing the events).

IV. Results

This section gives a full walkthrough of the ransomware from the initial infection of the user to the encryption of their files and to their subsequent decryption. Additionally, it shows how the monitoring software can detect when ransomware is likely being used.

To begin, the attacker needs to create a malicious image to send to the user. Due to the principle of open design, we can assume the attacker is aware of the source code of the image-viewing application and the operating system of some victim, and as such they can construct an input file that exploits vulnerabilities in their designs. The attacker first creates some malicious software that they want to be run on the victim's device, which in our case will be a mass-encryption tool described next. They can compile this software into an executable file, and then create an image with the qualities described in section III. Besides this, the attacker will additionally use the RSA key-generation program to create a `server-key.pub` and `server-key.pri`. The private key will be kept hidden by the attacker, and the public key will be embedded in the malicious executable file.

Next, the attacker can simply send the image to a victim which is known to utilize this image-viewing application. They can do this through email, forums, social media, or other image-sharing formats. It is important to note that the malformed nature of this image will likely prevent it from being rendered on some well-designed image-viewing applications, especially if they have measures in place to prevent malformed images from being displayed. Although, this is not always the case; as described in section III, the width, height, and image content does not need to be zero (they can be from a genuine image) as the malicious file is only appended to the end of the file for the purpose of replacing the log file generated by the image-viewing application itself. Therefore, some applications which ignore appended data can still render the image just fine, which makes this attack undetectable unless the user manually downloads the image onto their computer and views it through their own software.[3]

The toy image-viewing application made for this project works by (1) parsing an image in a standard image format with the `parse_image` program and (2) viewing the raw data of the parsed image with the `view_image` program. The vulnerability occurs when the `parse_image` program attempts to create a log file, which the malicious image overwrites with executable data with the same file name as `view_image`. Since the `view_image` program is executed afterwards, the malicious executable data is unknowingly run by the user and the attack proceeds.

After that, the cryptography model generates an RSA key-pair named `client-key.pub` and `client-key.pri`. The private key is encrypted using the `server-key.pub` embedded earlier, and the original key is deleted from the user's system. Now, the program can start using the client's public key to encrypt all the files in a given directory.[4] The only way to decrypt these files is to use the `client-key.pri` file which can only be obtained by the server that owns the `server-key.pri`.

Then, each encrypted file is passed through a small steganography program which pulls a random image from online[5] and encrypts the file entirely within it. The file is resampled with a point filter to fit the data as close as possible, noting that 4-bit LSB encoding means the size of the `ppm` file should be double that of the encrypted data file.

At this point, the user's files are encrypted. The user can navigate to the encrypted directory and find that files like `dissertation.tex` have been transformed into `dissertation.tex.enc.ppm`, which may depict something innocuous like a picture of a dog. An advanced user (or one with lots of money to spend on cryptography experts) may be able to retrieve their files without the use of a key by attempting a chosen plaintext attack to recreate their file based on the expected file format and contents due to the lack

[2]The significance of the constant 0.8 is to give a small amount of leeway for the delay in encrypting files. This allows one counter value to be slightly smaller than the other while still being relatively equal to it.

[3]In fact, the default image viewer for the Windows 10 operating system works this way for PNG files. Editing a PNG by appending random data will not prevent image-viewing applications from opening them, so our attack would at least be viable in this case.

[4]In our case, we only target the files contained in a `critical` directory on the user's desktop. This can be modified to, for instance, target all files from the user's root `/` directory, or to only target data files like `pdf`'s or `jpg`'s that are likely to be both small in size and important to the user.

[5]We used the attacker's server to do this for simplicity with a script that pulls a random `ppm` file from the home directory, but this could come from a website that returns random image files like `thispersondoesnotexist.com`.

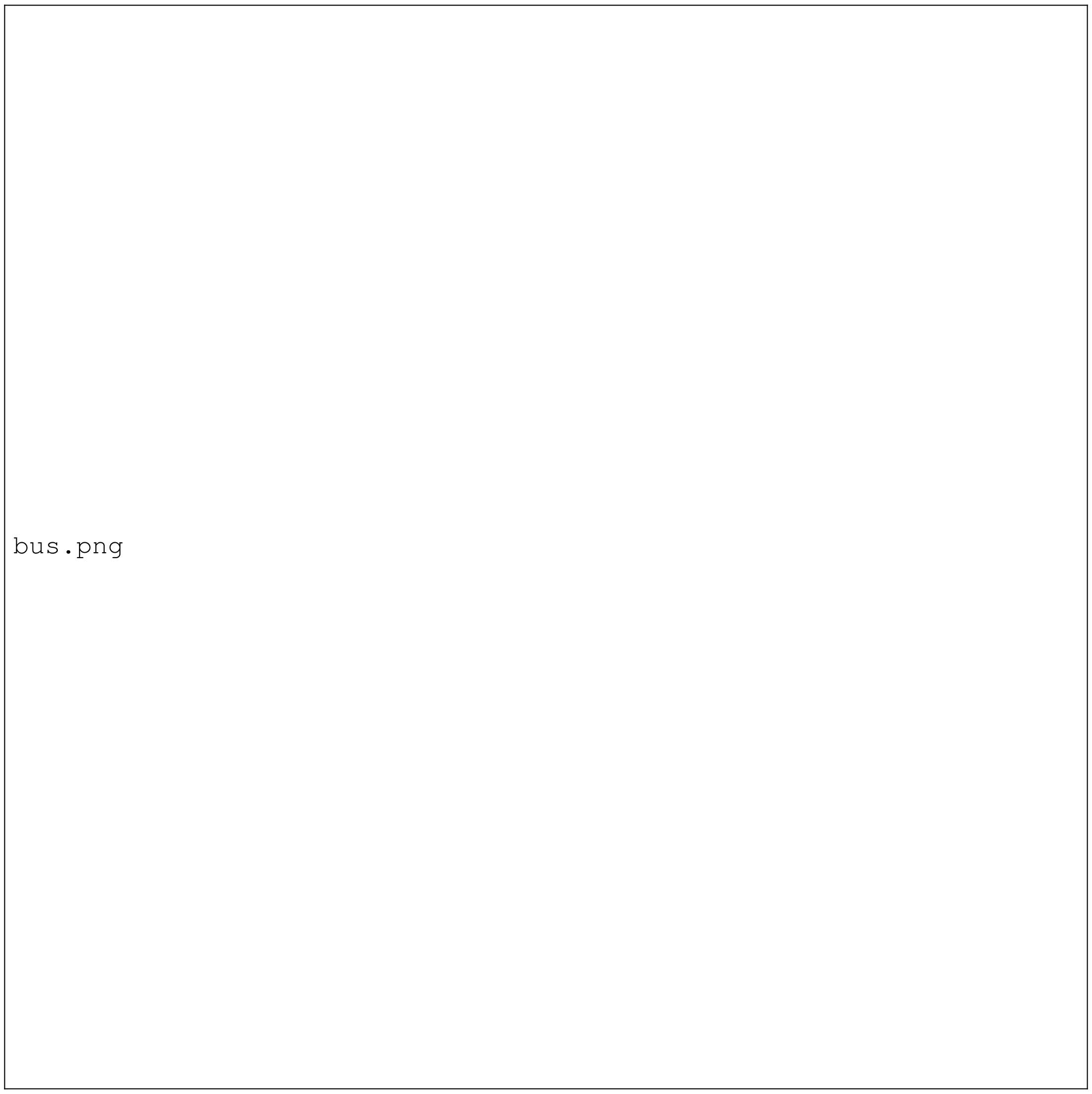

Fig. 2. Enlarged image comparing (left) a base image and (right) the base image with "lorem ipsum" sample text encoded at four LSB per byte

of RSA padding, but the average user cannot reasonably expect to discover this on their own.

Once the user decides to pay for decryption, they send both payment and their encrypted public key file `client-key.pri.enc` to the malicious server. The server then decrypts this file using their own private key `server-key.pri`, yielding a new `client-key.pri` that is sent back to the client. Finally, the client re-runs the image application with the modified `view_image` program, which checks if the decrypted client key exists in the same directory as the encrypted files. If it does, the program takes each file, scrapes the message from the LSB-encoded file, decrypts the encrypted data with the client's private key, and removes any trace the ransomware left on the user.

Lastly, the monitor can be run on the target `critical` directory to detect when the ransomware is being performed. When a directory initially has 8 files in it, this causes the `accdel_threshold` to equal $min[10, 0.8 \times 8]$, or 6. This means that when at least 6 accesses and deletions of the same file is performed, as well as at least $0.8 \times 6 = 4$ file creations are performed in the span of 10 seconds, then a ransomware is assumed to be taking place. When the monitor is running in the background, it successfully detects the execution of the ransomware.

References

[1] Handrizal, J. T. Tarigan, and D. I. Putra, "Implementation of steganography modified least significant bit using the columnar transposition cipher and caesar cipher algorithm in image insertion," in *Journal of Physics: Conference Series*, 2021, pp. 1–7.

[2] K. Conrad, "The Miller-Rabin test", *University of Connecticut*, 2017.

[3] H. Rowland, "The role of prime numbers in RSA cryptosystems", *Georgia College and State University*, 2016.

[4] E. Milanov, "The RSA algorithm", *University of Washington*, 2009.

[5] K. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities", in *Journal of Software Practice and Experience*, 2003, pp. 423–460.

[6] A. Smith and Y. Zhang, ”On the regularity of lossy RSA: Improved bounds and applications to padding-based encryption,” in *International Association for Cryptographic Research*, 2015, pp. 609–628.
[7] GMP MP: The GNU multiple precision arithmetic library.