CS424 Assignment 2

Jacob Honoroff, Benny Tsai

November 8, 2005

1 Kerberos Attack

- Attack plugin: KerberosEvilEchoClient.java
- Attack script: kerberos-crack.script
- Sample attack output: kerberos_attack.out

1.1 The adversary is a legitimate user of the system

The provided Kerberos implementation has a weak seeding for the Random Number Generator (RNG) that is used to generate the TGS and KDBM secret keys, as well as session keys. Due to its low amount of randomness, the adversary can brute-force a key by making a reasonable guess on certain parameters and then iterating over all possible values of the others, all on the order of minutes. We implemented an attack in which an evil client contacts the KDC to obtain a TGT, and then uses the TGT to obtain the K_{TGS}, with which the evil client may forge tickets for arbitrary services.

In the KDC's response to a client (message #2), a TGT encrypted under the TGS's private key is provided to the client. This ticket allows an adversary to verify guesses on the TGS's private key, since the ticket has multiple fields with values are known by the adversary (client name, client IP address, service name). To verify a guess, try to decrypt the encrypted TGT with the candidate key. The adversary then checks to see if the decrypted object is a proper ticket, with the correct values in fields with known value. If this is the case, then he knows that his guess was correct.

1.1.1 Entropy Overview

The entire entropy for seeding the RNG only comes from

1. A call to `System.currentTimeMillis()`
2. The difference between the time returned in step 1 and a previous call to `System.currentTimeMillis()`
3. A random value from 0-15 in the `randNode` calculation

4. Two bytes from the `getRandomBytes` that can take on one of 23 values and are heavily biased towards just a few values

The random number seeding has a bunch of other operations, none of which add to entropy. This includes taking the first eight bytes at the beginning of a random class file, which is the same for every class file: `0xcafebabe0000002e`.

For the TGS key, the the previous time call in step 2 was made during the KDBM key generation, which is only a difference of around 30 ms in practice.

In our attack, the user specifies the time they believe has elapsed since the TGS key creation until the time that they logged on. The attack runs over a range of 100 ms prior to this value up through this value. If it fails, the user is asked to try another value. In practice, our attack script has a delay of 500 ms from a `wait 500` call between TGS key creation and the attacker login, when we specify a range of 400-500 we always crack the key.

Interestingly, because the seed values are xor'd together, we can actually get lucky in the exhaust when wrong guesses of multiple values cancel each other out when xor'd together, (This was observed in once instance when a wrong guess of step 2 and step 4 canceled each other out).

1.1.2 A Close Look at `getRandomBytes`

The `getRandomBytes` function is a very convoluted way of accessing a certain field in each class file. From Sun's website[1], it appears that the value accessed is the class's "fields_count" which "gives the number of field_info structures in the fields table. The field_info structures represent all fields, both class variables and instance variables, declared by this class or interface type".

Regardless of what the value represents, the important thing is it is completely deterministic once the class file is chosen. Further, the possible values are heavily biased towards lower numbers, and out of all the class files in the classes directory there are only 23 unique values. The following chart shows these unique values it and the frequency for which they occur in the simnet classes:

```
0: 26
1: 19
2: 18
3: 12
4: 15
5: 5
6: 5
7: 3
8: 4
9: 1
11: 3
12: 2
```

[1] `http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html`

```
13: 1
14: 2
15: 1
17: 1
18: 1
21: 1
23: 1
26: 1
30: 1
32: 1
57: 1
```

So in our attack, we attempt to iterate through this list in an intelligent way. We make this exhaustion the outer loop for the attack, so that we can take advantage of the occurrences of the high-frequency bytes by trying guesses made from these bytes first in the attack. We thus order our exhaust over the two class file bytes by starting with the most likely tuples of these bytes, created by combining the highest frequency bytes.

The tuple order used is thus:

```
(0 0)
(1 0)
(0 1)
(1 1)
(2 0)
(0 2)
(2 1)
(1 2)
(2 2)
(4 0)
(0 4)
(4 1)
(1 4)
(4 2)
(2 4)
(4 4)
(3 0)
(0 3)
(3 1)
(1 3)
(3 2)
(2 3)
(3 4)
(4 3)
(3 3)
(5 0)
(0 5)
```

1.2 The adversary does not have legitimate access to the system

The problem is not much more difficult for an adversary who doesn't have legitimate access to the system. The adversary only needs to know a valid username, and he can send a message #1 masquerading as the user. He can then perform an off-line dictionary attack to recover the client password (and consequently the client key), verifying a guess by checking whether a decryption of message #2 has the correct nonce, which he chose in message #1. After he obtains the client key, he may decrypt message #2, extract the encrypted TGT, and perform the attack described in Section 1.1.

Even simpler, if he can eavesdrop and capture message #3, he can go ahead and carry out the attack on weak key generation without having to guess a client's password, because the encrypted TGT can be directly extracted from message #3.

2 Synkill

- Test script: synkill.script
- Expire timeout used: 5 sec
- Stale timeout used: 30 sec

An attacker can still SYN flood a target from a spoofed address simply by sending just two ACK packets before the flood. As long as the spoofed address is not EVIL, it can transition to GOOD from NULL or NEW with only 1 ACK packet, and it can transition from BAD to NEW with 1 packet, meaning it takes at most 2 packets to be classified GOOD. Note that Synkill does still help with syn floods by sending the immediate ACK so that the victim is not left in half-open states.

The problem is that Synkill does not keep track of TCP state. It tries to assume that certain traffic (ACKs and RSTs) which would occur from an address in the GOOD state making legitimate TCP traffic is in fact due to legitimate traffic coming from that GOOD address, even though it is easy for an attacker to spoof a few packets of this type without a connection ever having been made.

The fix is thus for an augmented FSM to keep track of TCP connection state when making decisions about the state of addresses. A transition into GOOD from NEW or NULL should only be made if the ACK or RST received was a legitimate response from a packet sent by the victim. Likewise, a transition from BAD to NEW should only be made on a legitimate ACK or RST, not simply any ACK or RST.

Synkill also exhibits other problems from not keeping track of TCP state. This includes that a GOOD address goes stale even if it is in the middle of a connection, and it will expire shortly thereafter, causing Synkill to reset a legitimate user. Note that there is not even an update of the staleness timeout when

ACKs are received in the GOOD state, meaning there is a fixed amount of time after the SYN in a legitimate connection after which the GOOD address will go stale, even if there are multiple packets exchanged throughout the connection. This renders Synkill useless with applications that involve long running connections, like ssh or ftp. A fix to the FSM should also keep addresses in the GOOD state while they have a live connection with the victim.