University of Maryland
CMSC498L — Cybersecurity Lab
Fall 2012

Midterm Exam — Sample Solution (8 questions)

1. (**28 points**) CTR-mode encryption of a message $M = m_1, m_2, \ldots$, where each m_i represents a 16-byte block, is done by choosing a random 16-byte IV and then outputting the ciphertext

$$IV,\ AES_k(IV) \oplus m_1,\ AES_k(IV+1) \oplus m_2,\ \ldots.$$

For encrypting messages whose length is not a multiple of the block length, assume padding is used exactly as in class. (So all messages are padded to be a multiple of 16-bytes long; if $W \in \{1, \ldots, 16\}$ bytes need to be appended then the W-byte string

$$\underbrace{W \ldots W}_{W \text{ times}}$$

is appended, with W represented in hex.) Assume decryption is done exactly as in class, so an error is returned if padding is not in the correct format.

Say you are given a 3-block ciphertext IV, C_1, C_2 encrypted as described above, and are additionally given access to a padding oracle (i.e., an oracle that returns 1 iff the input ciphertext decrypts to a properly padded message). Denote the ith byte of, e.g., C_1 by $C_1[i]$ (starting from 0).

   (a) How would you use the padding oracle to determine the length of the underlying plaintext (before padding)? Use pseudocode if that helps clarify things.

   Answer: (Note that decryption in CTR-mode is done *differently* from CBC-mode.)

```
for i=0 to 15:
    C2[i] = C2[i] ^ \0x01
    /* doesn't really matter what you do,
       as long as C2[i] is modified */
    submit (IV, C1, C2) to padding oracle
    if (padding oracle returns ``valid'')
       return 16+i
```

   (b) Assuming you have successfully accomplished part (a), how would you use the padding oracle to determine the final byte of the underlying plaintext (before padding)? Use pseudocode if that helps clarify things.

   Answer: In the following, IV, C_1, C_2 denotes the original ciphertext. Also, we let d denote the position of C_2 containing the final byte of plaintext, and let W denote the number of padding bytes appended.

```
for i=d+1 to 15:
    C2[i] = C2[i] ^ W ^ (W+1)
/* now the top-most W bytes will decrypt to W+1 */
orig = C2[d]
for i=\0x00 to \0xFF:
    C2[d] = i
    submit (IV, C1, C2) to padding oracle
    if (padding oracle returns ‘‘valid’’)
       return (W+1) ^ i ^ orig
```

2. (**15 points**) WEP uses RC4 for encryption, but RC4 has a number of known weaknesses that are exploited by the Fluhrer-Mantin-Shamir attack (and aircrack). Say WEP switched to AES, a secure block cipher that does not have any known weaknesses. Specifically, say the client and access point share a 128-bit key k, and encryption of a 128-bit packet m is done by choosing a random, 40-bit IV and sending

$$IV, \mathrm{AES}_k(IV) \oplus m.$$

(The IV is padded with trailing 0s to a length of 128 bits.)

(a) Is this a secure way to do encryption? Explain your answer.

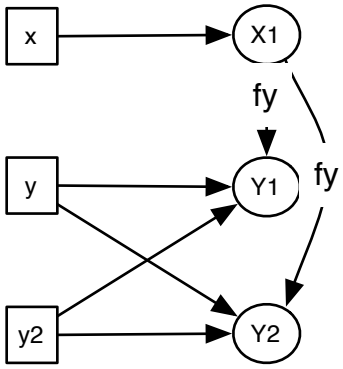
Answer: No. The problem is that the IV still repeats too frequently: a repeated IV is expected to occur after only 2^{20} packets are encrypted, and this allows an attacker to figure out some information about the encrypted packets. Even worse, an attacker may observe all IVs after a short amount of time, and (if the underlying plaintext is known for those packets) build a table of $(IV, \mathrm{AES}_k(IV))$ pairs.

(b) How would security be impacted if the IV were instead a 40-bit counter that was incremented each time a message was encrypted?

Answer: One could argue that it is slightly better, since repeated IVs now only occur after 2^{40} packets are encrypted. (This is still not good enough for adequate security, though.) One could argue that it is worse, because (assuming known plaintext is encrypted) the attacker now gets *all* $(IV, \mathrm{AES}_k(IV))$ pairs after 2^{40} packets are encrypted. Also, if the counter is reset upon initialization, low IVs will repeat very frequently.

3. **(14 points)** Consider the following Java program:

```
class X {
  private Y fy;
  Y get() { return fy; }
  void set(Y y) { fy = y; }
  void sink(Y p) { ... }
}
class Y {
  static Y source() {
    return new Y();
  }
}
class Main {
  static void main() {
    X x = new X();
    Y y = Y.source();
    Y y2;
    x.set(y);
    y2 = x.get();
    y = new Y();
    x.set(y);
    x.sink(y2);
  }
}
```



Solution to part (a)

(a) **(6 points)** Draw the points-to graph for this program for a flow-insensitive, context-insensitive analysis, as discussed in class. Recall that nodes in the graph are representations of local variables or objects, where for the latter there is one node that represents all objects allocated on a particular line of code. Edges represent "points-to" relationships: a node p has a directed edge to node q if p may point to q when running the program; when the source node is an object, the edge should be labeled with a field name.

> **Answer**: see diagram above. The variable `y` points to both `Y1` and `Y2` due to the two assignments, on lines 2 and 6 of `main`. Object `X1` points to these two objects via field `fy` due to the two assignments initiated by the calls to `set` on lines 4 and 7 of `main`. Finally, `y2` also points to both `Y1` and `Y2` because function calls are context-insensitive: since `fy` can point to both `Y` objects, setting `y2` via `x.get()` means `y2` could point to both. (Note you might have also added points-to edges for the parameters `y` and `p` in class `X`, but this was not required.)

(b) **(4 points)** Using a points-to graph, how does the LAPSE analysis described in the assigned reading tell whether an object originating at a particular source variable could end up at a particular sink variable?

> **Answer:** If the points-to sets of both the source and sink variables contain the object in question.

(c) **(4 points)** According to the flow-insensitive, context-insensitive points-to analysis, is it possible for `Y` objects produced by the `source` method to be passed to the `sink` method? If the analysis were flow-sensitive, but context-insensitive, would you get the same answer? What if it were both flow- and context-sensitive?

Answer: Yes. The `source` method in the above graph produces `Y1` objects (stored in variable `y`), and the argument to the `sink` method is `y2`, whose points-to set also includes `Y1` (because of the call to `x.set(y)`). The more precise (flow- and/or context-sensitive) analyses would similarly say there is a flow, though the points-to set for `y2` would be more precise (it would only point to `Y1`, which is the result of `source`, rather than both `Y1` and `Y2`).

4. **(7 points)**

Explain the difference between a *reflected* and a *persistent* cross-site scripting attack.

Answer: A reflected XSS attack is one in which the code in question is embedded within a link that the victim clicks on. This link takes the victim to the site that executes the embedded code. The attack on the `jobs.umd.edu` site mentioned in class was a reflected XSS attack. A persistent XSS attack is one in which the attacker is able to inject code on a web site that users subsequently access, and thus run with the privileges of the site. The Samy MySpace worm example mentioned in class is an example of a persistent XSS attack.

5. **(7 points)**

Suppose you know that a particular web site uses a backend database to implement authentication. Given a login page with *username* and *password* fields, what would you type into these fields to try to perform SQL injection to bypass proper authentication? Explain why your approach would work.

Answer: Authentication could be done by selecting a user's record from the database only if the password is correct. To perform an SQL injection in this case, you could type into the password field the text `' OR 1=1 --` . Or you could leave the password field blank and type in text `user' OR 1=1 --` into the user field (where `user` is the username you are interested in authenticating). In both cases, you end bypassing the password field. As an example, suppose the site constructs the query like so: `String query = "SELECT token FROM users WHERE user = '"+user+"' AND password = '"+pass+"'";`. If `user` and `pass` are the contents of the web fields then the resulting string will be `SELECT token FROM users WHERE user = 'bob' AND password = '' OR 1=1 --'` for the first form of the attack mentioned above. This will select Bob's token and skip the password check thanks to the added "or" clause.

6. (**15 points**) Write a complete C program such that, if you were to run the program under KLEE, then KLEE would explore exactly 6 paths through the program, and exactly 1 path would fail with an assertion error. You will probably find the following KLEE functions helpful:

   - `klee_make_symbolic(addr, num_bytes, name)`: store `num_bytes` bytes of symbolic memory at address `addr`.
   - `klee_assert(e)`: signal a failing execution if `e` evalutes to false.
   - `klee_assume(e)`: add `e` as an assumption to the current path.

   Answer: There are many possible answers. Here is one that works.

```
int main(void) {
  int a, b, c, x = 0;
  klee_make_symbolic(&a, 4, "a");
  klee_make_symbolic(&b, 4, "b");
  klee_make_symbolic(&c, 4, "c");
  if (a) x++;
  else x--;
  if (b) {
    if (c) x++ else x--;
  }
  else x--;
  klee_assert(x != 2);
}
```

7. (**7 points**) Explain what a *search strategy* is for a symbolic executor and why using one (or many) is necessary in practice.

   Answer: Symbolic executors explore many possible program paths, and the search strategy tells the symbolic executor how to prioritize its search. Search strategies are necessary because, in practice, symbolic executors cannot explore all program paths, hence they must instead try to explore the most promising paths within their time budget.

8. (**7 points**) List three commonly used approaches to protect against buffer overflows that smash the stack, and for each one briefly explain why they do not *guarantee* protection against overflows.

   Answer:

   - Stack canaries – the adversary may be able to overwrite the canary with the same value (perhaps using a format string vulnerability to discover what the value is)
   - Address space layout randomization – the adversary could still guess the starting address of stack, or could use other vulnerabilities to reveal it.

- Non-executable stack – the adversary could still use return-oriented programming to execute their desired code.