1. Yes, the value d will prove that the author of the joke was Joe and not Tom. As long as the secret key is kept secret, Lisa can be sure that Joe sent the message. The hash of the message that Joe sent and the hash of the message Lisa can compute must be the same in order for the message to be verified. If Tom messed around with the message, then its hash would change and Lisa would see that change to invalidate the message.
2. The attacker can see both X and Y being transferred by Alice and Bob. Both of these values contain the secret key within them, so the attacker can use XOR to find out the key. X xor Y is the same as X xor (Kb xor X). Using associativity, the expression turns into (X xor X) xor Kb. Kb is now isolated from the rest of the operation and revealed to the attacker.
3. 100^8 passwords. 100^8 times 10^{-9} seconds = 10000000 seconds to crack
4. The search space for the attack is 2^40*240,000 or 2.6388279e^17.
5. The size of Eve's search space for the attack is 2^32*500,000 or 2.1474836e^15. If Eve didn't have access to the file, she wouldn't know the salt length and the dictionary attack would take much longer since a different salt value would need to found for each userid.
6. Canary values only tell the operating system malicious code has overwritten parts of code that have or will execute, they do not prevent against attacks but notify an attack has occurred. A 10 bit canary will be sufficient in order to warn the operating system is comprised since it will be overwritten in the attack.
7. These benign viruses still may execute programs and take up space inside a computer, therefore they still have negative impact and should be removed.
8. The program to detect the virus would take C and xor it with the original program. If the result looks like one of the infected programs, then you can use the xor from problem 2 to isolate the key.
9. Keyloggers range from being incredibly simple to very complex. The safe answer would be to not enter the data. You could enter the text in the text window and then copy/paste the data into the web browser.
10. Shared secret keys need a key between each person who communicates with 100*99/2 keys or 4950 keys. For public key crypto you need one public key and one private key per person to make one key pair, so 100 people would need 100 keys.

The paper discusses the potential of a rule-based alternative to finding potential vulnerabilities in code samples containing buffer overflow errors. The ultimate result is the program BOTestGen which combines the rule-based outlook with other detection mechanisms like symbolic execution to locate code that result in buffer overflows. By using static analysis, the authors argue that the rule-based approach uses significantly less compute time and can be classified as “light-weight” compared to other approaches.

Approaching the problem, the authors of the paper utilize a convenient data structure to help the rules navigate to potential hits within the code. This data structure consists of a set of a nodes representing each vulnerable line of code in the program with input nodes that control the movement. Vulnerable lines of code include statements or functions that read/write into buffer spaces. The method then generates rules for these nodes by introducing several cases of character manipulation. Using these generated rules, the program can identify points in the code that will lead to buffer overflow.

Since the rule-based approach is static analysis, the rules work within small enough constraints to trigger exploits in the code instead of testing the constraints themselves. Therefore, the smallest input size is used throughout the program in order to create a buffer overflow. Typically, the rule-based approach uses an input size two or three times larger than the buffer size, but the technique also utilizes “existing symbolic evaluation” to find the least-size input to create buffer overflows.

The experiment setup for BOTestGen executes with another static analysis tool called CodeSurfer to check for C-based buffer overflows. The rules are generated using CodeSurfer’s API and the program uses the rules to generate input to find bugs within the application. The results show the source code locations in which the program is vulnerable to a buffer overflow attack and recorded for posterity.

At the end of the experimental round, the BOTestGen script was able to use several rules to execute bugs in known pieces of code that have buffer overflow attacks. In cases that the script was not able to locate bugs, the symbolic execution following rule-based transversal identified the buffer overflow attack.

In conclusion, the paper remarks positively about using rule-based approaches for detecting buffer overflows and proclaims the method can be used by developers with fewer resources than seeking out genetic algorithms or symbolic execution approaches. The rule-based approach takes in the smallest input and resources to find buffer overflow bugs, and thus can be light-weight compared to other methods. Using other methods are valid only for buffer overflows that are outliers or deal in the execution, non-static phase of a development-program lifecycle.