The Dark Side of "Black-Box" Cryptography or: Should We Trust Capstone?

Adam Young* and Moti Yung**

Abstract. The use of cryptographic devices as "black boxes", namely trusting their internal designs, has been suggested and in fact Capstone technology is offered as a next generation hardware-protected escrow encryption technology. Software cryptographic servers and programs are being offered as well, for use as library functions, as cryptography gets more and more prevalent in computing environments. The question we address in this paper is how the usage of cryptography as a black box exposes users to various threats and attacks that are undetectable in a black-box environment. We present the SETUP (Secretly Embedded Trapdoor with Universal Protection) mechanism, which can be embedded in a cryptographic black-box device. It enables an attacker (the manufacturer) to get the user's secret (from some stage of the output process of the device) in an unnoticeable fashion, yet protects against attacks by others and against reverse engineering (thus, maintaining the relative advantage of the actual attacker). We also show how the SETUP can, in fact, be employed for the design of "auto-escrowing key" systems. We present embeddings of SETUPs in RSA, El-Gamal, DSA, and private key systems (Kerberos). We implemented an RSA key-generation based SETUP that performs favorably when compared to PGP, a readily available RSA implementation. We also relate message-based SETUPs and subliminal channel attacks. Finally, we reflect on the potential implications of "trust management" in the context of the design and production of cryptosystems.

Key words: Cryptanalytic attacks, hardware, software, RSA, DSA, ElGamal, Kerberos, Private key, Public Key, applied systems, design and manufacturing of cryptographic devices and software, Capstone, key escrow, auto-escrowing keys, subliminal channels, randomness, pseudorandomness.

1 Introduction

Black-box cryptography (i.e., crypto using protected devices) is often used, and is strongly endorsed by the U.S. government, namely in the Clipper and in particular in Capstone escrow technology. Also, software cryptosystems are offered and used where users do not necessarily check their code authenticity. In effect,

* Dept. of Computer Science, Columbia University Email: ayoung@cs.columbia.edu.
** IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA. Email: moti@watson.ibm.com

N. Koblitz (Ed.): Advances in Cryptology - CRYPTO '96, LNCS 1109, pp. 89-103, 1996.

these modes employ cryptography as a black-box. Some of the most important questions that arise with respect to black-box cryptography are the following: Is the algorithm contained within the black-box secure? Does it leak secret key information? If someone ever successfully reverse-engineers the black-box am I at risk?

Here, we present the notion of a SETUP mechanism that allows implementors and attackers to modify cryptosystems so that they leak users' key information in such a way that protects the attacker and gives him (and only him) access to keys. In the case of black-box devices, our results show that the same exclusive ability of the attacker still holds in the case that a user successfully reverse-engineers the device. Thus, we shed some light on the above issues by showing how several existing cryptosystems can modified in this way. Furthermore, these modifications are internal, and are designed in such a way that the resulting cryptosystem conforms to the specifications of the original cryptosystem. Our results indicate that software cryptosystems can be modified to leak key information while significantly minimizing the attacker's risk of getting caught. This has serious implications for network security systems. When a cryptosystem is modified to leak secret key information subliminally using a SETUP implementation (rather than a standard one), we call the cryptosystem a *contaminated cryptosystem*. The modifications that we present are general in the sense that they can be implemented by the designer of a cryptosystem or by an attacker that uses rogue software.

Specifically, we first show how RSA and ElGamal key generation programs can be contaminated in such a way that a database of public keys created using the cryptosystem is effectively a database of public/private key pairs with respect to the attacker, exclusively. We also show how such an idea can be used for hardware based overhead-free "auto-escrowing key" systems. We then present message-based SETUPs that are related but stronger than the subliminal channels of Gus Simmons in ElGamal and DSA. We also show how Kerberos can be contaminated in such a way that the attacker can passively tap the network and exclusively derive session keys. We conclude with a discussion of suggested measures on how to reduce and detect the existence of contamination in various installations.

2 Definitions and Background

Informally, a Secretly Embedded Trapdoor with Universal Protection (SETUP) mechanism is an algorithm that can be embedded within a cryptosystem to leak encrypted secret key information through a subliminal channel in that cryptosystem. This encryption is performed by a PKCS function E that is also contained within the cryptosystem. Since the PKCS function E is a trapdoor one-way function, and since it is secretly embedded within the cryptosystem, we refer to the mechanism as a secretly embedded trapdoor. The information that is leaked through the subliminal channel of the cryptosystem is universally protected because even if the attacker is given access to the ciphertext and the embedded

function E, the secret key information still cannot be determined. The following is a more formal definition of a SETUP mechanism (contains all the details, but avoids lengthy formalisms so as to keep the idea clear).

Definition 1:

Let C be a publicly known cryptosystem. A SETUP mechanism is an algorithmic modification made to C to get C' such that:

1. The input of C' agrees with the public specifications of the input of C.
2. C' computes using the attacker's public encryption function E (and possibly other functions as well), contained within C'.
3. The attacker's private decryption function D is not contained within C' and is known only by the attacker.
4. The output of C' agrees with the public specifications of the output of C. At the same time, it contains published bits (of the user's secret key) which are easily derivable by the attacker but are otherwise hidden. (The output can be generated during key-generation or during system operation).
5. Furthermore, the output of C and C' are polynomially indistinguishable (see, e.g., [ACGS]) to everyone (including those who have access to the code of C') except the attacker.

Definition 2:

Let C be a publicly known cryptosystem. A contaminated cryptosystem C' is a modified version of C that contains a SETUP mechanism.

Thus by forming C' we have setup C to leak secret key information. Such an attack is carried out without letting the users know that the cryptosystem in question is contaminated, and without giving any advantage to those who discover the contamination.

Related Work

Gus Simmons has pioneered the research in the area of subliminal channels and their inclusion in cryptosystems [Sim85]. He has published channels in the Ong-Schnorr-Shamir, ElGamal, Esign, and DSA digital signature schemes. Another channel was discovered by Desmedt [Des90]. Killian and Leighton [KL95] showed how a key distribution channel containing a subliminal channel can be exploited by attackers that agree on a way to exploit it. We are inspired by all of these works and our goal is to point out that in the "black-box" context, the naive and somewhat restricted looking "imperfectness" exhibited by channels leaking information can become "serious flaws". Furthermore, even if all agreements among the parties are known, the danger persists due to "new" applications of cryptography itself (i.e., using crypto to attack crypto).

3 SETUP in RSA Key Generation

The obvious way to attack the RSA [RSA78] key generation process is to include a fixed prime number p. Since q will be chosen randomly, the modulus will look random to the casual observer. Obviously, this is not a SETUP since anyone can find all keys after two have been generated (using the Euclidean Algorithm). Hence, reverse engineering requires no effort at all. Using a fixed pseudorandom seed is easily detected by reverse engineering as well.

A more advanced mechanism is as follows. Let n be the product of two k-bit primes p and q. Let e and d denote public and private exponents respectively. In this description (e,n) and d are the keys being generated and (E,N) and D are the attacker's keys. The idea is to hide enough information within (e,n) to allow the attacker to derive d from (e,n). Assuming the original cryptosystem generates e at random from $\{0,1\}^k$, the following attack can be performed. p and q are chosen randomly from $\{0,1\}^k$ and are tested for primality. e is then set to be p^E mod N. Let this ciphertext be denoted by p'. If e and $\phi(n)$ are not relatively prime, a new value for p is chosen and the process is repeated. Once a valid e is found, d is computed as usual and the public key (e,n) is published. To determine p, the attacker looks up (e,n) and decrypts e with his private key. If the result divides n evenly then he has successfully factored the user's public modulus. This SETUP mechanism cannot be used effectively in programs like PGP, since PGP uses very small exponents (on the order of 5 bits). Therefore, any such attack is unlikely to go unnoticed. Also, attacks with small e will enable attacks on Rabin's scheme [Rabin].

We will now introduce our strongest version of the RSA SETUP mechanism by describing a program called Pretty-Awful-Privacy. PAP is very similar to PGP, except that the author of PAP has the exclusive ability to factor the public keys that are generated by PAP. In PAP, the problem of requiring a large exponent is circumvented entirely by hiding p' in the public modulus. PAP hides p' in the upper order bit representation of the public modulus, using a storage method for information within the RSA key that was first pointed out by Desmedt [Des90]).

PAP works as follows. It contains the k-bit RSA public key of the attacker which is half the length of the key being generated. It first generates a random k-bit prime p. It then randomizes p and makes sure that the resulting value is in the domain of the attacker's public key. Namely it "randomizes" p using a keyed randomizing function F. The key used in conjunction with F is $(K + i)$ where K is fixed and i is an index initially zero. Let the resulting value be denoted by p'. If p' isn't less than the attacker's k-bit modulus, then i is incremented and F is used once again. This process continues at most B_1 times. If after B_1 times, a value less than the attacker's modulus isn't reached, a new p is generated and the process is repeated (B_1 is, say, 16).

Once a value for p' is found it is encrypted using the attacker's public key to get p''. PAP then runs a pseudo-random function G on p'' using the key $(K + j)$ where K is the same as before and j is an index initially zero. Let the resulting value be denoted by p'''. Note that p''' is a k-bit quantity. (We tried

to achieve a pseudo-randomization of the values, and a mechanism to sample values relatively fast– the specific details can be changed). PAP then sets X to be p''' concatenated with a bit-string randomly chosen from $\{0,1\}^k$. To find the other factor q, PAP divides X by p and then tests the quotient for primality. q is set to be the quotient if and only if the quotient passes a primality test. If the quotient isn't prime, j is incremented by 1, and p''' is recomputed. PAP then reattempts to find a quotient that is prime. PAP will continue this process up to B_2 times. If this bound is reached, a new prime p is chosen at random, i and j are set to zero, and the entire key generation process is repeated. The B_2 bound gives a work factor that trades off the required work of finding the prime number q for increasing the work of recovering p [Has]. It can be shown using suitable values for B_1 and B_2 that the probability of finding a valid p and q is appreciable, using the Prime Number Theorem [And71]. PAP initially sets the value of the public exponent e being generated to 17. Once a valid q is found, PAP checks to see if e and $\phi(n)$ are relatively prime. If they aren't then e is incremented by 2 until they are. n and d are then calculated in the usual way.

To find out if a given public key was created using PAP, the attacker does the following. He first sets U to be the upper order bits of the victim's public modulus n such that there are k bits to the right of this value. He then decrypts U using $K+j$ and where j ranges from 0 to B_2-1. The attacker then decrypts all of these values using his private key to get the set of possible p'' values. Each potential p'' is then decrypted using F and $K+i$ where i ranges from 0 to B_1-1. If any of one of the resulting plaintexts divides n, then he has successfully factored the victim's modulus. If a factor isn't found, then the attacker decrypts $U+1$ and proceeds as before. Note that since PAP ignores the remainder upon dividing X by p, it is possible that a borrow bit modified p'' in the upper order bits of n. It is for this reason that the attacker must try $U+1$ as well. If by then, a factor isn't found, the attacker concludes that his version of PAP was not used to generate the public key.

Note that the reason for encrypting p with F prior to performing the public key encryption is to ensure that p can have a value larger than that of the attacker's public modulus! The reason for encrypting the public key ciphertext using G is to take advantage of the pseudo-randomness and to avoid the overhead of excessive public key encryptions. In doing these extra encryptions we cut down on the computational complexity of PAP and ensure the randomness of p and q. We implemented this SETUP mechanism using the GNU MP library. A more complete description of our implementation is given in Appendix A.

3.1 Security of PAP

We will now show that by making certain reasonable cryptographic assumptions, the values for p and q that are chosen by PAP are random. Note that p is contained in $\{0,1\}^k$, and that p is initially chosen uniformly at random. The randomizing function F is a mapping from the set of prime numbers in $\{0,1\}^k$ to [0..N-1], where N is the attacker's public modulus (recall that we only pass primes to F).

Lemma 1. *Assuming that p and the k upper order bits of X are random, q is random in the set of k-bit primes.*

Let C be an RSA cryptosystem that generates RSA public/private key pairs in the usual way, with the restriction that its random values are chosen independently from the user (this is the case in our modified version of PGP, see Appendix A). Now if we assume that the application of G is similar to applying a random oracle (which a pseudo-random function in hardware-protection is!) its range is indistinguishable from a truly random choice, hence we can show:

Theorem 1 *PAP is a contaminated cryptosystem based on cryptosystem C.*

Remark 1: The attacker's key is used as a private cipher (encryption key unknown– so a reduction of half the size is acceptable).

Remark 2: After reverse engineering, one learns the attacker's encryption key. If we assume it is a strong public-key system (given k) then the reverse-engineer cannot tell past or future keys since he does not see the random bits used in their generation. The "reverse engineer" still needs to solve strong encryptions by the attacker's key.

Remark 3: If we have the freedom to choose e, where e is half the size of n, then the attacker's key can be the same size as the keys generated by the system. In this case the encryption is split into two halves, half being put in N and half in e. In this case we can also use RSA as a strong encryption (pseudo-random generator [ACGS]), hiding the final seed for the attacker to invert in e.

4 An Application: Auto-Escrowing-Keys in Hardware

The notion of embedding a public key within a cryptosystem may lead to a globally trusted and efficient hardware key escrow mechanism. Each device would have its own unique public key. The corresponding private keys would be escrowed among two or more agencies (as in threshold cryptography and function sharing). If the communications from one device needs to be examined by law enforcement, the escrow agencies could combine their shares and the corresponding private key could be reconstructed. The communications device could be made tamperproof, and in the event that it is ever successfully reverse engineered, it will still be a difficult task to derive private keys. This would allow the general public to scrutinize the devices design, and would therefore provide assurance as to how it functions. Furthermore, there is no lengthy communications process between users and escrow agents in determining a key to be used, and the users are free to generate their own keys at any time. To ensure that users are using the escrow device to generate keys, the key distribution center can verify the SETUP existence before making keys publicly available. (A user may be required to perform key generation for a session based on its own and its partner's keys– this will be enough information for escrow regardless which partner is under a wiretapping procedure).

We have shown that under the hardware protection of key generation and assuming the use of RSA, we have, in effect, an escrow system. This is somewhat in the opposite direction of [BFL95] who showed that private encryption with universal escrow keys implies public-key cryptography.

Claim 1 *Given RSA (or a more general public-key function) with a SETUP in its key generation procedure, we can implement a tamper-proof hardware key escrow system with no system overhead.*

Warning: It could be the case that public escrow keys themselves get generated using a contaminated cryptosystem. The key escrow agencies would therefore be fooled into thinking they were the only people who could access the private escrow keys and guard the rights of individuals. So, "the guards themselves fail to guard". This hierarchy of attacks demonstrates the extreme level of caution that must be taken in regards to cryptosystems.

5 SETUPs in ElGamal, DSA, and Kerberos

SETUP in ElGamal Key Generation

A similar subliminal channel can be implemented in ElGamal. The following is a summary of normal ElGamal encryption [ElG85]:

Public Key: p,g,y
Private Key: x
Encryption: $a = g^k \pmod p$, $b = y^k M \pmod p$
Decryption: $M = b/a^x \pmod p$

Here M is the message being encrypted and (a,b) is the ciphertext of M. To generate a key pair, a prime number p is chosen at random (typically with known factorization [Bac88]). Two numbers, g and x are chosen at random such that they are both less than p and g is a generator. The value for y is then found by calculating $g^x \pmod p$. Two simple versions of the subliminal channel in ElGamal will now be described. Both versions require that the key generation program is capable of choosing x and either p or g.

In the first version, it is assumed that p is shared by a group of users and x and g are generated by the key generation program. This attack is very similar to the attack on RSA key generation. The value for x is chosen randomly and x is encrypted using the attacker's public key and a pseudorandom function to get x'. If x' is less than p and it is a generator mod p (e.g., assuming p's factorization is known) then g is set to x'. Otherwise, a new x is generated and the process is repeated. To retrieve x the attacker looks up the public key and decrypts g using his private key. The attacker's key is an RSA key, say.

Consider now the (less likely) case in which g is shared among a group of users and p is chosen by the key generation program. Another attack is as follows. The value for x is chosen randomly and is encrypted using the attacker's public key

and a pseudorandom function to get x'. If x' is a prime greater than g (and g is a generator mod x', e.g. assuming $x'-1$ has easy factorization into one large prime and other small primes), and x is less than x' , then p is set to be x'. The attacker can retrieve x by looking up the public key and decrypting p with his private key.

Pure ElGamal system: Consider the case where we are free to choose p and g during key generation. In this attack, x is encrypted using ElGamal rather than RSA. Since only ElGamal is used (and private key cryptography), the primitive routines for encryption need not be stored in the rogue routine since they are already present in the host cryptosystem. Let the attacker's keys be denoted by P, G, Y, and X. The contaminated cryptosystem generates x randomly and then computes g and p the following way. A value is chosen at random. If it is relatively prime to P-1, then k is set to be this value. b is then found by encrypting x with k, Y, and P using ElGamal. Hence $b = Y^k x \ mod \ P$. b is then encrypted with a private key to create pseudorandom functions and variability so that one option meets our required distribution (as in RSA, trying with increasing keys as a pseudorandom function untill a bound or a success is reached). If b is prime (we may require that the $b-1$-th factorization be known and have one large prime, for certifying the instance), and if x is less than b, then a is calculated using k, G, and P. Hence, $a = G^k \ mod \ P$. If a is not less than p then a new k is chosen and the process is repeated. Once a valid k, a, and b are found, p is set to b and g is set to a. (Recall that we may have to have special primes and a special generator according to the key generation procedure for the discrete logarithm problem in question.) Once g and p are chosen, y is then calculated using g, x, and p. If the user publishes y, g, and p, then the attacker can compute x by decrypting g and p with his private key.

The key generation attacks against RSA and ElGamal bear a strong resemblance to the ideas described in "Reflections on Trusting Trust" by Ken Thompson [Tho84]. Can programs be trusted to generate keys for us? Can the programs that make key generation programs be trusted? One way to prevent these attacks is to design key generation programs so that the user has the option to choose his or her own random parameters whenever possible, or to at least allow for testing at the time of installation (in hardware). The user should be able to check the devices manufactured by various vendors and compare the results. This would limit the avenues that an attacker could use to install a SETUP. It was pointed out to us by Diffie [Diffie] that in a typical cryptographic system, a key generation program is often put into hardware in order to be able to declare that "our system is secure". This "traditional wisdom" may need revision in light of the attacks presented herein. The trust between producers of cryptosystems and users has to be built on a different foundation.

SETUP in ElGamal Signature Scheme

In this section we introduce a SETUP where the leaking is done via the system's messages (i.e., signature values). The attack is general in the sense that users can change their public/private key pairs at any time and the attack will

still work. Note that in the subliminal channel attack on signature schemes by Gus Simmons, the attacking parties collaborate, namely, Bob must know Alice's private key in order to receive a subliminal message. This is not the case in the SETUP attack.

Let p and g (for the signature alg.) be shared among the users. A rogue routine is installed in ElGamal that contains the ElGamal public key of the attacker. Let the public key of the attacker be denoted by p, g, and Y and let his private key be denoted by X. Note that the g and p in the attacker's public key are the same as those in the ElGamal implementation. Let the user's private key be denoted by x.

Our attack continually leaks x such that only the attacker can retrieve it. For the attacker to derive x he must obtain at least two (wlog, consecutive) signatures from Alice (at some point during the signing history), denoted by (r_i, s_i) and (r_{i+1}, s_{i+1}). It is also assumed that none of the random parameters are disclosed to the user, to assure that the user cannot detect an attack. The computation of the signature (r_i, s_i) proceeds in a similar way as in normal ElGamal. A random number k_i is generated such that k_i and $p-1$ are relatively prime. In addition, k_i is used iff $\gcd(Y^{k_i} \bmod p, p-1) = \gcd(g^{(Y^{-k_i} \bmod p)} \bmod p, p-1) = 1$. The signature of Alice's message m_i is found by calculating, $r_i = g^{k_i} \bmod p$, and $s_i = ({k_i}^{-1}(m_i - xr_i)) \bmod p-1$. Alice's subsequent signature is determined in a slightly different way than usual. Rather than choosing k_{i+1} randomly, its inverse is chosen to be a specific value. We set k_{i+1}^{-1} to be $Y^{k_i} \bmod p$. k_{i+1} is then found by calculating the inverse of $k_{i+1}^{-1} \bmod p$. The signature algorithm then proceeds as normal. We set $r_{i+1} = g^{k_{i+1}} \bmod p$ and $s_{i+1} = ({k_{i+1}}^{-1}(m_{i+1} - xr_{i+1})) \bmod p-1$. Given these two digital signatures and the corresponding messages, the attacker can derive x by computing, $x = r_{i+1}^{-1}(m_{i+1} - (s_{i+1}/({r_i}^X \bmod p))) \bmod p-1$. Since x was chosen to be less than $p-1$, this yields Alice's private key. Furthermore, no one else can compute x since no one else knows the private key X.

Theorem 2 *Given r_i, r_{i+1}, s_{i+1} and m_{i+1}, the attacker can compute x.*

Proof.

$$s_{i+1} = (Y^{k_i} \bmod p)(m_{i+1} - xr_{i+1}) \bmod p-1$$

$${r_i}^X \bmod p = g^{Xk_i} \bmod p = Y^{k_i} \bmod p$$

$$s_{i+1}/({r_i}^X \bmod p) \bmod p-1 = (m_{i+1} - xr_{i+1}) \bmod p-1$$

$$r_{i+1}x = (m_{i+1} - (s_{i+1}/({r_i}^X \bmod p))) \bmod p-1$$

$$x = r_{i+1}^{-1}(m_{i+1} - (s_{i+1}/({r_i}^X \bmod p))) \bmod p-1$$

Comment: the probability of getting two consecutive signatures that permit the computation of x can be increased by being more selective of the k_i. The following is how to accomplish this. We make k_{i+2} a function of k_{i+1} in the same way we made k_{i+1} a function of k_i. We then make k_{i+3} a function of k_{i+2}, and so on. We therefore include a pseudo-random number generator for the k_i in the

contaminated cryptosystem. The effectiveness of this method is limited due to the restrictions on k_i. We can reduce this drawback by "looking ahead" and only using k_i's that yield sets of valid k_i's with high cardinality.

The attack has been inspired by the work of Gus Simmons. The attack is unique in that it exclusively allows an attacker to compute x based on information arranged by the cryptographic device. This attack, which is quite simple to implement, implies that there may exist other SETUP attacks on cryptosystems that give the implementor exclusive access to all enciphered information.

Application: Recall now the hypothetical situation proposed by Gus Simmons regarding his subliminal attack on ElGamal. Alice is in prison and wants to coordinate an escape plan with Bob, who is on the outside. It was originally assumed that Bob already knows Alice's private key. The SETUP attack is more general since we can discard this pre-coordination assumption: Alice need only look up Bob's public key, and then contaminate her own cryptosystem with it. She can then send two signed messages to Bob, thereby giving him her private key. The Simmons subliminal channel can then be used as usual. If their relationship ever goes awry, Alice can rekey and seek a new person to help her plan an escape. In general this can be phrased as:

Theorem 3 *If a system has a message-based SETUP version, and the users are members of a public key system (based on a trapdoor permutation like RSA), then there exists a subliminal channel between users who have not met earlier.*

SETUP in DSA: SETUP from subliminal channels

In fact, the concept of securely disclosing keys via a SETUP can be used to extend subliminal channels in general (ignoring speed and bandwidth), e.g. the one found in DSA by Gus Simmons [Sim94]. One of the shortcomings of the attack on DSA is that only a few bits (roughly 14) can be leaked in a given signature. The other drawback is that if anyone successfully reverse-engineers the tamperproof device, they will have access to the secret primes. The later drawback can be readily solved by including the attacker's public key in the encryption device as a SETUP, and having the device compute the encryption of the user's private key with the attacker's public key, with the ciphertext bits leaked in the same way as described by Simmons. This will prevent everyone except the designer of the device from being able to derive the private keys of others. A tamper proof device is therefore not needed, and the set of secret primes can be made public. This is a general strengthening, which gives the following implication (that is somewhat converse to the last theorem):

Theorem 4 *If a cryptosystem has a subliminal channel, then assuming a trapdoor permutation (RSA, say), it has a message-based SETUP version.*

SETUP in Kerberos

In this section we show how Kerberos [NT94] can be modified to leak session keys exclusively to an attacker without putting the attacker at risk. The Kerberos

model is based on a client server model in which the client is either a user or a program. Upon logging in, the user first communicates with the Kerberos authentication server and receives a ticket granting server (TGS) ticket. This ticket is used to receive subsequent tickets to be used with various servers. Once the user decides which service he wants, he sends a ticket request along with the TGS ticket to the ticket granting server. The user then receives a ticket for a particular server. All Kerberos tickets have time-stamps and are only valid for a specified time interval. A concise description of the ticket granting server interaction will now be given.

Kerberos Table of Abbreviations

c : *client*
s : *server*
k_x : *x's secret key*
$\{m\}k_x$: *m encrypted with x's secret key*
$T_{x,y}$: *x's ticket to use y*
$A_{x,y}$: *Authenticator from x to y*

The following applies to Kerberos Version 5. To receive a server ticket and server session key, the client sends the packet $(s, \{T_{c,tgs}\}k_{tgs}, \{A_c\}k_{c,tgs})$ to the TGS. If the TGS ticket and Authenticator are valid then $(\{k_{c,s}\}k_{c,tgs}, \{T_{c,s}\}k_s)$ is sent back to the client.

Claim 2 *Based on any public-key cryptosystem there is a SETUP version of the Kerberos key distribution mechanism.*

The SETUP involves modifying the way the TGS functions by including a rogue routine and the attacker's public key. Rather than generating $k_{c,s}$ randomly, the rogue routine receives $k_{c,tgs}$ from the ticket and creates the plaintext m using $m = (k_{c,tgs}, RND)$, where RND is a random field. The plaintext m is then encrypted using the public key of the attacker to get the ciphertext m' (say of 512 bits). Since m' is longer than the ticket we have to split it among a number of tickets. m' is exposed block by block, each time the value of $\{k_{c,s}\}k_{c,tgs}$ is set to be the next (still unexposed) block of m'. Each of the $k_{c,s}$'s is then found by decrypting the corresponding block of m' using $k_{c,tgs}$. The derived $k_{c,s}$ is then placed into the $T_{c,s}$ ticket. Given the blocks of m' and the private key of the attacker, one can get m' and recover the desired key, $k_{c,tgs}$.

In more detail, this attack will give the attacker access to the values $k_{c,tgs}$ and $k_{c,s}$ that can be found by passively tapping the network. The attack proceeds as follows. The attacker modifies Kerberos to become a contaminated Kerberos by modifying the Ticket Granting Server. He then eavesdrops on the network and picks up packets emanating from the TGS. He also records any communications session that he desires. When he wants to decrypt the session of client c (assuming the session was SETUP), he performs the following algorithm. First, he decrypts enough ciphertexts $\{k_{c,s}\}k_{c,tgs}$ using his private key to get the blocks of the plaintext m. From m he gets $k_{c,tgs}$.

The attack is secure against system administrators who discover the contamination. The attacker breaks the symmetry between what he can see and what the system administrator can see by including a public key within the TGS. Without the private key, the set of $\{k_{c,s}\}k_{c,tgs}$'s cannot be used to get $k_{c,tgs}$ by anyone except the attacker.

6 Conclusion

SETUP attacks would completely compromise system security if they were implemented and would give a unique advantage to the attacking party. Fortunately, these mechanisms can only be abused by powerful entities, those who implement systems and those who have root access to software. These attacks require one-time access to software or devices. These attacks also have serious implications for smart card technology. Should we trust the key generation software that comes with a smart card? Even if key generation software is digitally signed we have no assurance that it wasn't contaminated by its implementor without explicitly analyzing its code. The SETUP system "looks just the same". This is a serious problem, particularly if the software is proprietary and incorporates anti-piracy mechanisms to make analysis difficult. Capstone, cryptographic servers, and cryptographic libraries are all guards used to prevent system infiltration. Due to the existence of SETUP attacks, measures need to be taken to guard these guards. We conclude with recommendations that will help eliminate or minimize the effects of the attacks.

1. Control of randomness is important given its indistinguishability from pseudorandomnes. Thus, design software and hardware that permit the user to choose random parameters, and make the algorithms used publicly known. This allows the user to compare the output of one implementation with the output of a trusted implementation, based on user supplied parameters, which should be the same.
2. If software is used to generate keys, be absolutely certain that the software is trustworthy. Integrity checks can help detect modifications made to software after installation.
3. Cascading cryptosystems that are designed and implemented by independent sources is also a good measure.
4. In the case of smartcards, make the card support third party random number generation devices. This will help convince users that a SETUP mechanism isn't being used in the smartcard.
5. Make sure the randomness source, the key generator, and the user (message supplier) are three system components which are separated but are well authenticated, hard to bypass, and have private channels between them to assure secrecy.
6. Industry standards for testing-modes which work with user supplied randomness should help increase trust in hardware devices.

One problem is that very often when left to their own devices, users do not choose truly random numbers. Yet our results indicate that cryptosystems cannot be trusted to do so either. It may therefore be desirable to have a separate program or hardware device that generates random values. It is also obvious that the "common wisdom" of reducing lack of trust to a "hardware component" with a well defined specification, needs revision. The hardware components and their source have to be included in the trust model of the system.

In summary, we presented the notion of a SETUP mechanism and showed attacks against RSA, ElGamal, DSA, and Kerberos. The attacks employed "crypto from within" to attack cryptographic systems. We believe that it is important for designers and system administrators to be aware of the potential of attacks like the ones described herein. By taking appropriate measures, analyzing trust relationships, and by making the necessary modifications to existing systems, we can try to ensure that cryptosystems provide the degree of trust and security that we expect them to provide.

Acknowledgements:
We would like to acknowledge the help of Matt Hastings for refining some of the attacks described.

References

[ACGS] W. Alexi, B. Chor, O. Goldreich and C. Schnorr. RSA and Rabin Functions: Certain Parts are as Hard as the Whole. In *SIAM Journal of Computing*, volume 17, n. 2, pages 194–209, April 1988.

[And71] G. E. Andrews. "Number Theory," page 100, 1971. Dover Publications Inc.

[Bac88] E. Bach. How To Generate Factored Random Numbers. In *SIAM Journal of Computing*, volume 17, n. 2, April 1988.

[BFL95] M. Blaze, J. Feigenbaum and F.T. Leighton. Masterkey Cryptosystems, CRYPTO 95 Rump session, Aug. 1995.

[Des90] Yvo Desmedt. Abuses in Cryptography and How to Fight Them. In *Advances in Cryptology—CRYPTO '88*, pages 375–389, Berlin, 1990. Springer-Verlag.

[Diffie] W. Diffie, Personal Communication.

[DSS91] Proposed Federal Information Processing Standard for Digital Signature Standard (DSS). In volume 56, n. 169 of *Federal Register*, pages 42980–42982, 1991.

[ElG85] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology—CRYPTO '84*, pages 10–18, Berlin, 1985. Springer-Verlag.

[Has] Matthew B. Hastings, private communication.

[KL95] J. Killian and F.T. Leighton. Fair Cryptosystems Revisited. In *Advances in Cryptology—CRYPTO '95*, pages 208–221, Berlin, 1995. Springer-Verlag.

[LMS] J. Lacy, D. Mitchell, W. Schell. CryptoLib: Cryptography in Software. AT&T Bell Laboratories, section 2.2.1.

[MB95] D. Mitchell, M. Blaze. truerand.c, AT&T Laboratories, 1995.

[NT94] B. C. Neuman, T. Ts'o. Kerberos: An Authentication Service for Computer Networks. In *IEEE Communications Magazine*, pages 33–38, Sept. 1994.

[Rabin] M. Rabin. A Public-key and Signature Scheme as Secure as Factoring, MIT Tech. Report, 1978.

[RSA78] R. Rivest, A. Shamir, L. Adleman. A method for obtaining Digital Signatures and Public-Key Cryptosystems. In *Communications of the ACM*, volume 21, n. 2, pages 120–126, 1978.

[Sim85] G. J. Simmons. The Subliminal Channel and Digital Signatures. In *Advances in Cryptology—EUROCRYPT '84*, pages 51-57, Berlin, 1985. Springer-Verlag.

[Sim94] G. J. Simmons. Subliminal Channels: Past and Present. In *European Trans. on Telecommunication*, 5(4), 1994, PAGES 459-473.

[Tho84] K. Thompson. Reflections on Trusting Trust. In *Communications of the ACM*, volume 27, n. 8, August 1984.

[WN] D. Wheeler, R. Needham. Tiny Encryption Algorithm (TEA). In *Fast Software Encryption: second internation workshop*, volume 1008 of Lecture Notes in computer science, Dec. 1994. Springer.

[Zim92] Phil Zimmerman. *PGP User's Guide*, 4 Dec. 1992.

A Comparative Performance: RSA SETUP vs. PGP

We compared the average key generation running time of our "SETUP program" with a modified version of PGP 2.6. Our program was written in ANSI C and was linked with the GNU MP library version 1.3.2. Our program generates a 512 bit RSA public/private key pair using the SETUP mechanism described in this work. Our implementation uses truerand [MB95], which is part of CryptoLib [LMS], to generate physically random seeds for the pseudo-random number generator. We chose to use TEA [WN] as our pseudo-random function (any other block cipher like DES will do). We used the probabilistic primality test from Knuth to test the random values. We found that we had good results with B_1 equal to 16. The value for B_2 was 512.

Our goal in doing the comparison was only to see if our RSA SETUP mechanism took noticeably more time than PGP, and to get a feel for the practicality of the SETUP as a solution to the problem of key escrow. Since our program was developed using the GNU MP library, and since PGP is based on RSALIB, we did not do as close a comparison with PGP as possible (since we wanted only rough figures). Ideally one would start with PGP and then modify it as little as possible in order to introduce a SETUP mechanism. Our (quick) approach was to modify PGP to use the same random number generation routines, and to make it generate primes in a similar manner as the SETUP.

The primary changes that we made to PGP were the following. We modified randombits() to invoke rand() instead of randomunit(). We removed the PGP random generation routine calls in rsa_keygen(). We also removed the test that is performed on the new key. We modified randomprime() to be the following:

```
int randomprime(unitptr p, short nbits)
{
int   numTested=0;
```

```
GenThatPrime:
if (numTested == 10)
  {numTested = 0;printf("Testing 10 more nums for primality\n");}
srand(truerand());randombits(p,nbits-2);numTested++;
mp_setbit(p, nbits - 1);mp_setbit(p, nbits - 2);
if (primetest(p))  return 0;
else goto GenThatPrime;
}                                   /* randomprime */
```

We performed our benchmark from the beginning of rsa_keygen() up until the end of rsa_keygen().

Table 1
512 bit RSA key generation times in seconds

Trial	Modified PGP	SETUP gen	SETUP decr
1	94	94	9
2	104	136	92
3	100	215	23
4	157	153	21
5	114	20	2
6	132	173	47
7	79	127	25
8	76	274	10
9	158	40	10
10	75	69	20
Average	108.9	130.1	25.9

The Modified PGP column lists the modified PGP key generation times. The SETUP gen column lists the SETUP key generation times. The SETUP decr column lists the amount of time required to derive a private key from the corresponding public key. We found that there is no appreciable difference between the running times of the modified PGP and our SETUP. We therefore believe that it may be possible to modify PGP to contain an RSA SETUP mechanism such that it can't be detected by analyzing key generation times alone.