Attacks on Message Stream Encryption [⋆]

Billy Bob Brumley and Jukka Valkonen

Department of Information and Computer Science
Helsinki University of Technology
P.O.Box 5400, FIN-02015 TKK, Finland
{billy.brumley,jukka.valkonen}@tkk.fi

Abstract. Message Stream Encryption (MSE) provides obfuscation, data confidentiality, and limited authentication to BitTorrent clients. Although obfuscation of header and payload data was the main design goal of MSE, users understandably still expect data confidentiality and authentication from their BitTorrent clients. In this paper, we present numerous attacks on the MSE protocol itself, independent of clients. We then test many popular BitTorrent clients for vulnerability to these attacks, resulting in a number of serious vulnerabilities in popular clients. These results are timely and significant due to the high penetration rate of BitTorrent clients.

Key words: BitTorrent, peer-to-peer protocols, stream ciphers, man-in-the-middle attacks.

1 Introduction

In 2004, former CacheLogic now Velocix, a Cambridge, UK company, conducted a six month study in which they concluded that BitTorrent traffic accounted for roughly 33% of all Internet traffic [1]. Although such estimates vary and should be taken with a grain of salt, the proliferation of BitTorrent clients cannot be denied.

As such usage has an adverse affect on network performance, some ISPs have chosen to throttle BitTorrent traffic. In response to this, BitTorrent client developers designed Message Stream Encryption (MSE) [2]. The main design goal of MSE was indeed traffic obfuscation, with secondary goals of confidentiality and authentication.

Surprisingly, instead of using an existing, well-known public protocol as a basis (IPSec, for example), BitTorrent client developers themselves designed the MSE protocol from scratch. To this effect, in this paper we present a number of attacks on the MSE protocol, with vulnerabilities leading to complete key recovery and torrent fingerprint leakage. We apply these attacks to a number of individual clients for many different platforms. The results show that the MSE protocol has a significant number of weaknesses and the protocol description itself leaves too many details up to interpretation.

2 The BitTorrent Protocol

BitTorrent [3] is a protocol designed to distribute large files efficiently across networks. The idea of the protocol is to reduce the cost and burden of a single device when large files are being distributed. This is achieved by distributing the file into multiple small pieces which are then supplied by multiple clients. While devices download files using the BitTorrent protocol, they simultaneously start sharing the same pieces.

BitTorrent network consist of four entities: downloaders, seeders, web servers and trackers. In order to start sharing a file using the protocol, first a torrent file is created. The file includes all information needed to download all the pieces of the shared file. Basically, this means tracker information, the total amount of pieces the file is shared into and hash values of all the pieces. These hash values are used to verify the integrity of the file after the pieces are downloaded. In

[⋆] Proceedings of the 13th Nordic Workshop on Secure IT Systems—NordSec '08 (Oct 9-10, 2008, Copenhagen, Denmark), pp. 163–173.

addition, the file includes a hash file called InfoHash which is computed from the other metadata in the torrent file and thus can be used to identify the specific torrent. In addition to giving the torrent file for users to download, a torrent tracker needs to be informed about the file. The basic task of the tracker is to keep track of all the seeders of a file and thus help the downloading clients in communication with each other.

The torrent file is uploaded on a web server to be distributed to downloaders. When a user would like to download a file using BitTorrent, he first contacts the web server to download the torrent file. (Message 1 of Figure 1). The file can be downloaded in a number of ways, normally via HTTP or HTTPS. After downloading the torrent file, the BitTorrent client contacts the specified tracker to get the current list of seeders seeding the file to be downloaded. (Message 2 in Figure 1).

After receiving the list of seeders, the client contacts them one at a time (Figure 1: 3a,3b,3c) to download a specific piece of the file the seeder is providing. After downloading a piece (or multiple pieces), the client is able to verify the integrity of the file using the information from the torrent file.

If compared to downloading by traditional means, the speedup of BitTorrent protocol comes from spreading the file over multiple devices. All peers in the BitTorrent network need to seed only small pieces over multiple TCP connections, whereas in traditional web downloading the large files are distributed by single serves over single TCP connections. In addition, in HTTP downloading the file is downloaded in sequential manner, whereas in BitTorrent the pieces are downloaded in randomized order.

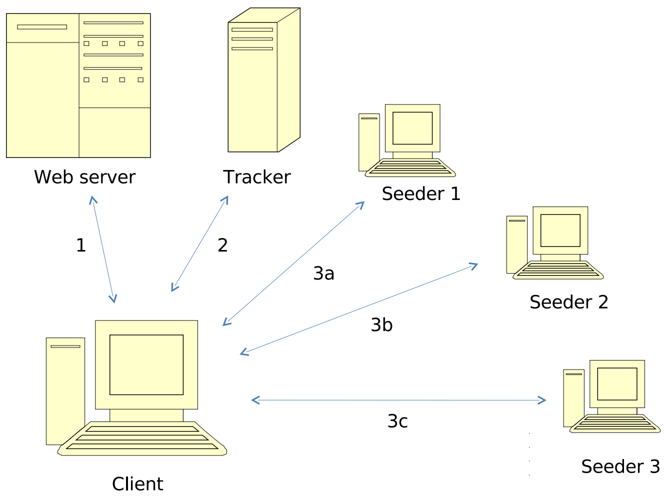

Fig. 1. BitTorrent operation

2.1 Torrent Handshake

We briefly describe the handshake of the BitTorrent protocol itself [3]; this takes place in the layer above MSE. The handshake is the first message transmitted between clients and is generally used to ensure that two clients are indeed about to share the same file. The handshake is 49+PStr bytes long and consists of the following fields (in order):

- PStrLen: 1 byte. The length of the PStr field.
- PStr: PStrLen bytes. Protocol identifier; for torrents, "BitTorrent protocol" and PStrLen=19.
- Reserved: 8 bytes. Initially all-zero, used to modify the protocol behavior.
- InfoHash: 20 bytes. The SHA1 hash of the info key in the metainfo file; unique identifier for a single torrent.

– PeerID: 20 bytes. A unique string identifier for the client. In practice, two bytes are used to identify the client software, four bytes for the client software version, then random numbers.

If a seeder client receives a handshake with an InfoHash they are not serving, they would be expected to drop the connection.

3 Message Stream Encryption

Message Stream Encryption [2] is designed to provide security features to the BitTorrent protocol at a lower layer by acting as a wrapper. The main goal of the protocol is to provide obfuscation for the data streams. This is done to prevent passive eavesdroppers from being able to recognize the protocol that is used. Recently, this has become an issue as some Internet service providers have started to monitor the data streams and in some cases limit the bandwidth of clients using BitTorrent.

In addition to obfuscation, secondary goals of the protocol appear to be providing some level of confidentiality and authentication of the communicating peers; this is evident from the cryptographic primitives being used.

The handshake of the protocol is depicted in Figure 2. The protocol uses Diffie-Hellman key exchange [4] to create a session key (surprisingly, not one of the numerous standard methods for authenticated key exchange [5–7] since a pre-shared secret is available). In addition, a weak shared key is used to provide some level of authentication to the key exchange. The InfoHash value transmitted in the torrent file is used for this weak shared secret, in this case it's value being similar to a pre-shared key in the context of peer-to-peer networks.

After negotiating the Diffie-Hellman key, the rest of the handshake is encrypted using RC4 [8], with the first 1024 keystream bytes discarded to defeat the attacks in [9, 10]–clearly some level of confidentiality (say, against those not participating in the torrent) is desired. The RC4 key is based on both the Diffie-Hellman key and the weak shared secret, thus a traditional man-in-the-middle attack will not be successful without the weak shared secret. The payload is either encrypted using RC4 or sent in plaintext. The desired payload delivery method is negotiated in Messages 3 and 4. In Message 3, the initiator sends the method(s) it is willing to use in CryptoProvide and the responder selects the desired method in CryptoSelect.

Note also that MSE was designed as a general payload delivery protocol, not specific to BitTorrent. When looking for weaknesses in MSE, it is thus important to consider what the intended payload is. Weakness in MSE need not be connected to BitTorrent in any way.

3.1 Security and MSE

The security of peer-to-peer networks like BitTorrent inherently has different goals than other networks like GSM or wireless LANs. The driving force behind P2P networks is the openness and ability to distribute and connect. In fact, the original creator of the BitTorrent protocol, Bram Cohen, was opposed to the widespread adoption of MSE, concerned that it would cause too much incompatibility between clients.

We quote directly from the MSE protocol specification [2] describing the design methodology of MSE:

> It is also designed to provide limited protection against active MITM attacks and portscanning by requiring a weak shared secret to complete the handshake. You should note that the major design goal was payload and protocol obfuscation, not peer authentication and data integrity verification. Thus it does not offer protection against adversaries which already know the necessary data to establish connections (that is IP/Port/Shared Secret/Payload protocol).

– g: Generator; with MSE $g = 2$ generates $\mathbb{Z}_p^\times$ with 768-bit safe prime $p = 2q + 1$.
– a, b: Diffie-Hellman exponents, MSE recommends 160 random bits.
– S: Negotiated Diffie-Hellman key, $\mathsf{S} = (g^a)^b = (g^b)^a = g^{ab}$.
– PadA, PadB: Random padding of length 0–512 bytes.
– H: Hash function, SHA-1 which outputs 20 bytes. Literal strings are enclosed in ' '.
– SKey: Weak shared secret key; for BitTorrent, SKey = InfoHash of size 20 bytes.
– E: Encryption, RC4 with key H('keyA',S,SKey) for A and H('keyB',S,SKey) for B.
– $\hat{\mathsf{E}}$: Negotiated encryption; currently either plaintext or the continuation of the above encryption.
– VC: Verification constant, currently 8 bytes of `0x00`. Used for synchronization.
– CryptoProvide: 4 byte field denoting which encryption primitives are supported; currently `0x01` = plaintext and `0x02` = RC4. Supported methods are combined with XOR.
– CryptoSelect: 4 byte field denoting the chosen encryption primitive, using the same values as above.
– len: Length function, outputs the byte length of the input as 2 bytes.
– PadC, PadD: Zero-valued padding of 0–512 bytes.
– IA: Initial payload; for BitTorrent, A's portion of the BitTorrent handshake.

1. A $\rightarrow$ B: g^a, PadA
2. B $\rightarrow$ A: g^b, PadB
3. A $\rightarrow$ B: H('req1', S), H('req2', SKey) $\oplus$ H('req3', S), E(VC, CryptoProvide, len(PadC), PadC, len(IA)), E(IA)
4. B $\rightarrow$ A: E(VC, CryptoSelect, len(padD), padD), $\hat{\mathsf{E}}$(Payload Stream)
5. A $\rightarrow$ B: $\hat{\mathsf{E}}$(Payload Stream)

Fig. 2. Message Stream Encryption Protocol handshake

We consider what the protocol states it is designed to protect against. The payload protocol is clearly the BitTorrent Peer Wire (TCP) protocol in this case. To carry out an attack, an attacker would presumably have to know (or have a way to get) the target connection details such as IP and port. Thus the only seemingly interesting case to attack, and for very logical reasons, is when an attacker does not know the shared secret. This shared secret is considered "weak" as the privacy differs greatly from what would normally be expected of a shared secret. In this case, the weak shared secret being the InfoHash of the file means that if you know what torrent you want to participate in, you are able to, and thus such an attacker is free to unleash a plethora of attacks. Such a simple protocol cannot hope to protect against this.

Thus, in this paper we restrict to the case where the attacker does not know the weak shared secret. This is not at all unreasonable, as tracker data can (and for MSE to provide confidentiality, clearly should) be exchanged over SSL, so the simple attack target would be unveiling the data being shared through a seemingly secured BitTorrent protocol using MSE.

We are thus not attacking in any way the obfuscation portion of MSE. The attacks considered here should be considered targeted attacks that tell something about the payload. This would not be useable by an ISP on a large scale to throttle BitTorrent traffic, but could be used to target a specific user. For example, a law enforcement agency might want to know what payload a user is transmitting or receiving (say, for copyright infringement issues). A user would reasonably expect MSE to be able to defend against such attacks.

4 Message Stream Encryption Weaknesses

In this section, we present some major weakness of MSE, independent of the client implementations. These attacks are applied to a number of BitTorrent clients in Section 5.

4.1 Lack of Message Authentication

As there is no message authentication present on the messages sent in the MSE handshake, attackers are free to modify any of the messages sent in an undetectable manner. With RC4 being a stream cipher, it becomes incredibly simple for attackers to make predictable changes in resulting plaintext after decryption. We now show how attacker can modify the messages sent in the MSE handshake to trick the devices into downgrading the encryption into plaintext, leading easily to recovery of the weak shared secret (SKey, InfoHash).

The stream cipher RC4 produces ciphertext by combining a pseudorandom sequence of bytes with plaintext using the XOR operation. The attack works simply by flipping bits of the CryptoProvide field by using the XOR operation on the ciphertext, as shown in Figure 3. In order to be able to modify the fields in practice, the attacker needs to be able to synchronize with the handshake and locate the CryptoProvide value in Message 3 of Figure 2 to get the devices to downgrade the encryption. An attacker can normally accomplish this as the messages of the handshake are usually sent in separate packets. After the attacker has located the correct message, finding the CryptoProvide field is easy: the lengths of the fields before the desired fields are known from the specification. The attacker then makes the desired changes using the XOR operation, and is able to downgrade the encryption from RC4 to plaintext. For a client desiring the highest level of security, only a value of CryptoProvide=0x02 would be sent, meaning the client only provides RC4. The limited number of possible values makes it simple for an attacker to achieve the desired effect through trial and error.

A → B:	H('req1', S)	H('req2', SKey) ⊕ H('req3', S)	VC	CryptoProvide	...
	20 bytes	20 bytes	0000000000000000 ⊕ $b_0 \dots b_7$	00000002 ⊕ $b_8 \dots b_{11}$	...
Attacker:	↓	↓	↓	⊕ 00000003	...
B:	—	—	⊕ $b_0 \dots b_7$ = 0000000000000000	⊕ $b_8 \dots b_{11}$ = 00000001	...

Fig. 3. Encryption downgrade attack on MSE.

Having now received a CryptoProvide value of plaintext only, B responds with Message 4 of Figure 2. The last portion of this message is the payload in plaintext, which is B's portion of the BitTorrent protocol handshake as described in Sec. 2.1; this includes the weak shared secret (SKey, InfoHash). Having recovered the weak shared secret, the attacker now knows what torrent is being shared and can, for example, run a traditional man-in-the-middle attack to recover actual bytes being sent with the torrent.

It is also entirely possible that B rejects providing plaintext only. However, as the protocol does not provide any kind of message integrity checking, it is up to the implementation of client B to drop the connection if plaintext is selected even though encryption is required. Results on implementations are described in Section 5.

Note that such bit flipping as described above is also possible with the CryptoSelect field in Message 4 of Figure 2. However, the attacker is less likely to be successful in this case; the method of encryption has already been chosen by B in the last portion of Message 4 and will be evident when received by A, who will be expecting B's portion of the BitTorrent protocol handshake.

The attack can be prevented quite easily. The protocol should implement some kind of integrity checking. This could be achieved using for example Message Authentication Codes (MAC) (see for example [11] or Cyclic Redundancy Check (CRC) (e.g. [11]) techniques. As the protocol already uses a (weak) shared secret, implementing a method based on a MAC would be quite easy. For example Wired Equivalent Privacy (WEP) [12] uses RC4 cipher together with Cyclic Redundancy Check CRC-32.

4.2 Keystream Leakage

Many of the fields in the MSE handshake have a large number of bytes but take either one or a small number of values. Ordinarily, this would not be a significant problem; however, the zero-valued variable length padding causes immediate leakage of a significant amount of keystream bytes. Again, this does not seem like a big issue–unless the keystream is reused. Below we present an attack that exploits the zero-valued padding when the keystream is reused.

In two runs of the MSE handshake for the same torrent, consider two values of Message 3 of Figure 2 denoted c_1 and c_2 using the same keystream. The length of PadC is a random value from 0 to 512, thus with all probability one of c_i is longer than the other; assume without loss of generality that c_1 is longer than c_2, and we denote the length of the PadC values as i and j respectively. The MSE protocol description states of PadC and PadD: "For padding-only usage in the current version they should be zeroed." We examine the construction of the c_i in Figure 4. We can see from the very first message that because of the fixed values of many of the fields, we recover many keystream bytes b, and with $j < i$ most noteably $b_{14+j} \dots b_{14+i-1}$. With all probability, this leads to the instant recovery of IA from c_2 by just XORing the fixed known keystream bytes. Again, this initial payload consists of A's portion of the BitTorrent protocol handshake, and thus contains the weak shared secret (SKey, InfoHash) which can be recovered as long as $j + 48 \leq i$ (see Sec. 2.1). This already holds with high probability for two sessions, but can be repeated as many times as necessary to obtained the required result.

	VC	CryptoProvide	len(PadC)	PadC		len(IA)	IA
$c_1 =$	0000000000000000	00000002	????	00 ... 00	00 ... 00		
	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	...	...
	$b_0 \dots b_7$	$b_8 \dots b_{11}$	$b_{12}b_{13}$	$b_{14} \dots b_{14+j-1}$	$b_{14+j} \dots b_{14+i-1}$		
$c_2 =$	0000000000000000	00000002	????	00 ... 00		????	?? ...
	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$	$\oplus$
	$b_0 \dots b_7$	$b_8 \dots b_{11}$	$b_{12}b_{13}$	$b_{14} \dots b_{14+j-1}$		$b_{14+j}b_{15+j}$	$b_{16+j} \dots$

Fig. 4. MSE Secret key recovery attack by keystream reuse.

Forcing Keystream Reuse in Practice With the attack in Figure 4, the attacker plays the role of B and thus would need some way to force reuse of the keystream. At first glance, the above attack does not seem very practical as the only way to force keystream reuse is when the same RC4 key is reused. Above, it was assumed that the same torrent was being accessed in both runs of the MSE handshake; thus, the only way the keystream would be reused is if the negotiated Diffie-Hellman key S was the same in both runs. We give two examples of how an attacker might accomplish this; results of running these attacks on different BitTorrent client implementations are given in Section 5.

Keystream reuse using trivial subgroups. With $p = 2q+1$ a safe prime, the only small subgroups of $\mathbb{Z}_p^\star$ are the trivial ones; that is, $\{1\}$ of multiplicative order 1 and $\{-1, 1\}$ of order 2. The Diffie-Hellman key can be confined to one of these trivial subgroups by raising the public keys sent to the power of $2q$ or q, respectively [13]. A simple example of this attack follows. A attempts to open a connection to B; the attacker is in the middle and impersonates B, receives Message 1 and sends the public key value 1 in response, records Message 3 from A and drops the connection. The process is repeated again, either for the same A or different A. The Diffie-Hellman key $\mathsf{S} = 1$ is the same in both runs of the MSE handshake, and the attack in Figure 4 is most likely successful. Unlike the MSE specification, most standards explicitly state to validate public keys; for example, the Wireless USB association models specification [14].

Keystream reuse using discrete logs. Keystream reuse can also be forced if computing discrete logs of the public keys sent in the Diffie-Hellman key agreement is possible. For example, A sends $g^{a_1} \mod p$ and the attacker sends $g^{b_1} \mod p$, records Message 3 from A and drops the connection. Now A sends $g^{a_2} \mod p$. Assume the attacker can somehow compute a_1 and a_2. The Diffie-Hellman negotiated key for the first session was $\mathsf{S}_1 = g^{a_1 b_1} \mod p$. To force keystream reuse in the second session, the attacker chooses b_2 such that $a_1 b_1 \equiv a_2 b_2 \pmod{\operatorname{ord}(g)}$ holds. When the attacker sends $g^{b_2} \mod p$ in the second session, the negotiated key $\mathsf{S}_2 = g^{a_2 b_2} \mod p = \mathsf{S}_1$ and the same keystream will be used in the second session. The textbook example of how computing discrete logs could be easy for a given client implementation is poor pseudo random number generation—for example, seeding with a 32-bit value or repeated seeding based on the system clock or CPU counter.

4.3 Torrent Fingerprint Leakage

As the protocol uses the weak shared secret SKey as part of the encryption key, the devices communicating must be able to negotiate which key to be used. Clients seed many torrents concurrently, thus determining which SKey to use in MSE is tricky. When the communication is initiated, the devices presumeably share the same SKey, the seeder simply does not know which key the downloader is using (and thus what torrent is being accessed). In the current implementation, the client tells the seeder which key to use in Message 3 in Figure 2 by sending a hash of the SKey; we denote this as the *torrent fingerprint*. An active man-in-the middle is able to obtain this fingerprint using XOR as it knows the negotiated Diffie-Hellman key S. As there is no randomness added to this hash, the torrent fingerprint (as the name implies) is unique per torrent; a good analogy is to that of comparing salted password hashes (e.g. classical Unix password authentication) to password hashing with no salting.

The fingerprint leakage may lead to recovery of the SKey and/or the actual file being downloaded. As the SKey values are distributed within the torrent files, which are distributed to clients using normal web servers, the attacker can, after seeing a hash of some SKey, browse through the Internet for torrent files and compute SHA-1 hashes of them offline. At some point, the attacker may find the same fingerprint it has already seen. This could be considered a brute-force attack if torrents are taken at random, or a dictionary attack if an attacker has a fixed set of torrents they wish to check the torrent fingerprint against. In either case, the keyspace is very limited and all the keys are distributed on the Internet. For example, the administrator of a torrent website serving torrent files could easily, with one SQL query, generate all the torrent fingerprints of every torrent available from its server (and even publish this list). Adding per-session randomness to this hash could prevent such an attack (at the significant cost of seeding client performance), but still does not prevent the calculation of the hash values online even when randomness is included (although this is somewhat less of a threat).

This leakage also leads to the following security issue: even though the attacker is not able to directly tell which torrent is being downloaded, it can say if the torrent is the same as downloaded by another client in another instance as the SKey and thus torrent fingerprint is unique for each torrent file.

The situation is illustrated in Figures 1 and 5. In Figure 1 the client is downloading pieces from multiple seeders. As for all these seeders the InfoHash is the same and the same hash is sent in messages 3a, 3b and 3c, the attacker is able to tell that in all these connections the same file is being downloaded. Figure 5 depicts a situation where two separate swarms have been created. If the attacker is active in at least one pair of devices sharing and downloading a piece of a file in both of the swarms, it is able to tell if the same file is being downloaded by simply comparing the torrent fingerprints.

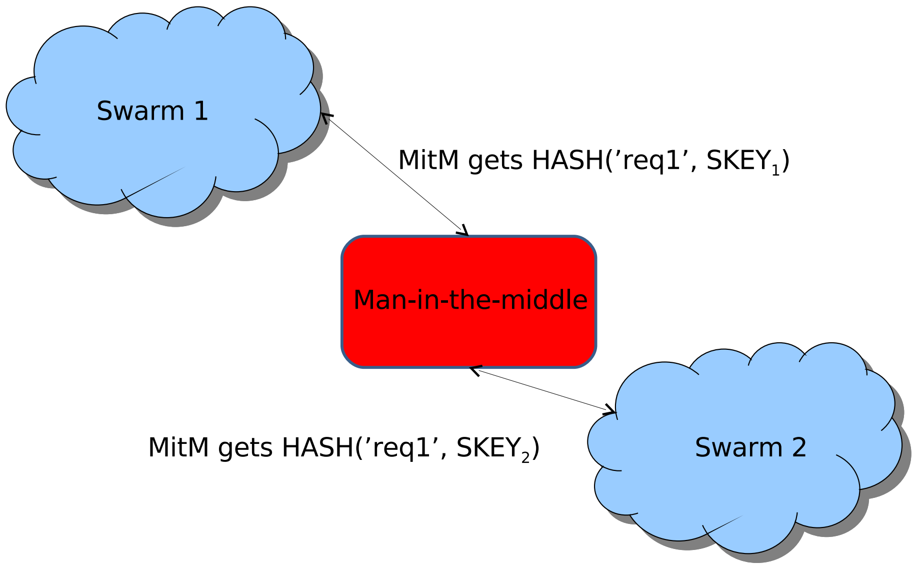
Swarm 1
MitM gets HASH('req1', SKEY$_1$)
Man-in-the-middle
MitM gets HASH('req1', SKEY$_2$)
Swarm 2

Fig. 5. Torrent fingerprint by man-in-the-middle in two swarms.

5 Vulnerabilities on Implementations

As we were interested in seeing how the real implementations handle the previously described security weaknesses, we implemented our own client capable of acting as both a client and a server in MSE. This was done as many BitTorrent clients are closed source; even with open source, understanding the implementation (and its flaws) can be quite difficult. By implementing a custom client, we were able to obtain full control and a clear idea of how different clients have implemented MSE. Next, we list some properties of tested clients.

5.1 Popular Clients

In 2006-7, Digital Music News conducted a study on the penetration of individual BitTorrent clients [15]. Statistics were collected from users voluntarily using an online virus scan site. Over 1.7 million PCs were examined. Table 1 (left) lists the results; (right) shows P2P software marketshare. The data was gathered by PC Pitstop as late as January 2008 [16], where over one million PCs were examined. We can conclude that μTorrent has by far the greatest market penetration for a BitTorrent client, at least for Windows.

Rank	Client	Installations (%)
1.	μTorrent	5.56
2.	BitTorrent	2.28
3.	Azureus	2.11
4.	Bitcomet	1.89
5.	Bitlord	1.27

Client	Marketshare (%)
LimeWire	37.19
μTorrent	13.51
BitTorrent	5.31
Ares	4.35
Emule	3.69
BitComet	3.64
Azureus	3.05
Other	29.21

Table 1. Popularity of BitTorrent clients.

The clients we tested for vulnerabilities are listed below. From this list, the degree of support for different MSE options ranges greatly.

BitTorrent 6.0.2 (Build 8388). Windows, closed source. This is the "official" BitTorrent client. Through Options, Preferences, BitTorrent, there are three options for Protocol Encryption, Outgoing: Disabled, Enabled, and Forced. There is a checkbox for "Allow incoming legacy connections".

μTorrent 1.7.7 (Build 8179). Windows, closed source. Through Options, Preferences, BitTorrent, there are three options for Protocol Encryption, Outgoing: Disabled, Enabled, and Forced. There is a checkbox for "Allow incoming legacy connections".

BitComet 0.99. Windows, closed source. Under Options, Options, Advanced, Connection, there are three options for Protocol encryption: Auto Detect, Always, and Disable.

Azureus 3.0.5.0. Multi-platform, Java, open source. Through Tools, Options, Connection, Transport Encryption, there are two options for Minimum encryption level: Plain and RC4. "Require encrypted transport" enables and disables MSE. There are two checkboxes for fallback options to allow incoming and/or outgoing unencrypted connections, as well as an option to support the cryptoport tracker extension.

KTorrent 2.1. Linux, C/C++, open source. Through Settings, Configure KTorrent, General, under Encryption there are two checkboxes for "Use protocol encryption" and "Allow unencrypted connections".

5.2 Results

A summary of the attack results is given in Table 2; we provide details of the individual attacks below.

Encryption downgrade. We tested the clients in two directions: the custom client acting as party A (initiating connections, client) and as party B (receiving connections, server). In both scenarios, the real client was set to allow only encrypted (i.e. RC4) connections. Our custom client then attempted to either provide or select (ignoring the other party's CryptoProvide field) plaintext only; according to the MSE specification, it is up to the client to drop the connection in these cases, although for this attack scenario, as previously mentioned it would be much more convenient to prevent the attack with the use of a MAC. The clients μTorrent and BitTorrent were found to be vulnerable; we were unable to find a difference between the "Enabled" and "Forced" for the "Outgoing" option of Protocol Encryption. KTorrent, Azureus, and BitComet all rejected selecting an unsupported encryption method, and hence were not found to be vulnerable.

Keystream reuse, trivial subgroups. To test for this vulnerability, three different values were sent as the public key to the real clients: values of 1, $p-1$, and 0. If the MSE handshake was successful, forcing keystream reuse via subgroup confinement is possible for that particular client. As the MSE specification does not specify to drop connections when receiving such values, it is up

Client	Encryption downgrade	Keystream reuse, subgroups	Keystream reuse, discrete logs	SKey Recovery	Fingerprint leakage
BitTorrent	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
μTorrent	$\surd$	$\surd$	$\times$	$\surd$	$\surd$
BitComet	$\times$	$\times$	$\times$	$\times$	$\surd$
Azureus[a]	$\times$	$\times$	$\times$	$\surd$	$\surd$
KTorrent	$\times$	$\surd$	$\surd$	$\surd$	$\surd$

Table 2. Summary of client vulnerabilities; $\surd$ =Vulnerable, $\times$ =Not Vulnerable.

[a] In sessions initiated by an Azureus client (e.g. acting as A), the BitTorrent handshake is sent in plaintext first—thus the shared secret is instantly revealed; we cannot explain such behavior.

```
BigInt BigInt::random()
{
static Uint32 rnd = 0;
if (rnd % 10 == 0) // reset every 10 calls
{
        TimeStamp now = bt::GetCurrentTime(); // seeding on system time
        srand(now); // 32-bit seed
        rnd = 0;
}
rnd++;
Uint8 tmp[20];
for (Uint32 i = 0;i < 20;i++)
        tmp[i] = (Uint8)rand() % 0x100;
return BigInt::fromBuffer(tmp,20);
}
```

Fig. 6. KTorrent 2.2.7, excerpt from `bigint.cpp` for random number generation.

to the implementation (thus these keys are valid according to the MSE specification, but it is still entirely possible that the implementation has foreseen such attacks). Many cryptographic packages perform such public key validation automatically, and a client that does so is clearly less likely to be vulnerable. Only two of the clients examined are open source; KTorrent's public key portion of the MSE implementation is done using a simple BigInteger package, and thus it is easy to identify the vulnerability. The clients μTorrent, BitTorrent, and KTorrent were all found to be vulnerable, while Azureus and BitComet both dropped the connection.

Keystream reuse, discrete logs. As mentioned above, only two of the clients are open source; we only examined these two clients for this particular attack. Azureus uses the canned Java methods for random number generation, hence we did not attempt any attacks on computing discrete logs with Azureus. However, KTorrent uses the standard C `srand` function to seed the pseudo-random number generator (PRNG). The seed provided by the current system time, reset every 10 calls. The function `srand` casts the provided seed as a 32-bit integer, and therefore it has a total entropy of $10 \cdot 2^{32} \approx 2^{35}$; see Figure 6, with the vulnerabilities highlighted in red. An attacker can easily compute all of these values offline, or attempt to run a more intelligent attack by guessing the seed based on the time.

Torrent fingerprint leakage. As this vulnerability is inherent to the MSE protocol, any client implementing MSE is clearly vulnerable.

6 Conclusion

During the past few years, the BitTorrent protocol has gained a large interest between users of the Internet for payload and content delivery, accounting for a significant portion of network traffic. The MSE protocol was designed to obfuscate such data, as well as provide limited confidentiality and authentication between peers.

In this paper, we have identified a number of significant weaknesses in the MSE protocol. We emphasize that these protocol weaknesses are not useful for ISPs wanting to throttle BitTorrent bandwidth on their networks, but lead to frighteningly feasible targeted attacks by ISPs or law enforcement agencies to reveal seemingly confidential data. We have shown that MSE does not provide a good defense against such targeted attacks.

At a minimum, we recommend the following changes to the MSE specification:

1. Add a Message Authentication Code (MAC) to the messages;
2. Drop connections if public keys (say K_A) do not satisfy $1 < K_A < p - 1 \pmod p$;

3. Remove or randomize padding that is encrypted, specifically the PadC and PadD values.

A solution to the torrent fingerprint leakage needs further investigation, taking efficiency, practicality, and security into account.

In conclusion, MSE still does a good job at data obfuscation as a whole. However, users expecting anything other than obfuscation, such as data confidentiality or authentication, cannot rely on MSE as the attacks presented here have shown.

Acknowledgments

The experiments appearing herein were carried out during the course “Special Course in Practical Security of Information Systems” at the Telecommunications Software and Multimedia Laboratory (TML), Helsinki University of Technology. We thank those that provided feedback, in particular the responsible lecturer Sami Vaarala.

Additionally, we thank Kaisa Nyberg for comments.

References

1. Andrade, N., Mowbray, M., Lima, A., Wagner, G., Ripeanu, M.: Influences on cooperation in bittorrent communities. In: P2PECON ’05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems, New York, NY, USA, ACM (2005) 111–115
2. MSE: Message stream encryption protocol. Azureus Wiki (January 2006) `http://www.azureuswiki.com/index.php/Message_Stream_Encryption`.
3. Cohen, B.: The BitTorrent protocol specification (January 10, 2008) `http://www.bittorrent.org/beps/bep_0003.html`.
4. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Information Theory **IT-22**(6) (1976) 644–654
5. IEEE: Standard specifications for password-based public-key cryptographic techniques. Draft IEEE P1363.2 / D26, Institute of Electrical and Electronics Engineers, Inc. (September 2006)
6. Bellovin, S.M.; Merritt, M.: Encrypted key exchange: password-based protocols secure against dictionary attacks. IEEE Computer Society Symposium on Research in Security and Privacy (4-6 May 1992) 72–84
7. Jablon, D.P.: Strong password-only authenticated key exchange. SIGCOMM Comput. Commun. Rev. **26**(5) (1996) 5–26
8. Thayer, R., Kaukonen, K.: A stream cipher encryption algorithm ’arcfour’. Internet Engineering Task Force (July 1999) `http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt`.
9. Mantin, I., Shamir, A.: A practical attack on broadcast RC4. In: Fast Software Encryption. Volume 2355 of Lecture Notes in Comput. Sci., Springer Berlin / Heidelberg (2002) 87–104
10. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the key scheduling algorithm of RC4. In: Selected areas in cryptography. Volume 2259 of Lecture Notes in Comput. Sci. Springer, Berlin (2001) 1–24
11. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1997)
12. IEEE Computer Society: IEEE Standard for Information Technology - Telecommunications and information exchagne between systems- Local and metropolitan area networks- Specific requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications `http://standards.ieee.org/getieee802/download/802.11-2007.pdf/`.
13. van Oorschot, P.C., Wiener, M.J.: On Diffie-Hellman key agreement with short exponents. In: Advances in cryptology—EUROCRYPT ’96. Volume 1070 of Lecture Notes in Comput. Sci. Springer, Berlin (1996) 332–343
14. WUSB: Association Models Supplement to the Certified Wireless Universal Serial Bus Specification - Revision 1.0 (2006) `http://www.usb.org/developers/wusb/wusb_2007_0214.zip`.
15. Digital Music News: Digital media desktop report, third quarter (2007)
16. TorrentFreak: Filesharing report shows explosive growth for uTorrent. `http://torrentfreak.com/p2p-statistics-080426/` (April 2008)